FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

Wrappers for Tracing Collective Communication Functions with PAT

Bart Theelen

FZJ-ZAM-IB-9811

August 1998

(letzte Änderung: 12.08.98)

Abstract

Effectively and efficiently implementing parallel programs for computer systems with a large number of processing elements involves the investigation of occurring communication between processes. The communication entails messages that are passed between participating processing elements. Analysing, debugging and tuning of parallel programs is very complex due to the dynamic behaviour of the communication.

Recording the communication events during the execution of a parallel program together with a time stamp assists a programmer in analysing the parallel program by enabling the reconstruction of the dynamic behaviour of the communication. An automatic instrumentation process changes the parallel program for recording all important events, including the communication events. In association with Cray Research the Zentralinstitut für Angewandte Mathematik at the Forschungszentrum Jülich implements an instrumentation module for the Performance Analysis Tool (PAT) for the Cray T3E parallel computer systems.

Next to Point-to-Point communication, the collective communication functions of the message passing libraries available on the Cray T3E parallel computer system enable communication involving more than one sending and receiving processing element. This report discusses the special wrapper functions used for instrumenting these communication functions.

Although three general forms of collective communication are available, four possible logical patterns exist. The final implementation of the wrapper functions require adaptation of the general frameworks that record a specific logical pattern for a collective communication function. The correctness of the implemented wrapper functions that are discussed in this report is founded by a series of test programs.

Investigating the internal implementation of the collective communication functions of the message passing libraries results in the conclusion that several collective communication functions could be implemented more efficiently. A special set up is used to analyse the internal implementation of the collective communication functions with PAT.

Together with PAT, the wrappers for the Point-to-Point and collective communication functions will soon be available in the ‘CrayTools’ module of Cray Research.

Contents

Page:

1 A brief Overview of the Company **1**

2 Introduction **3**

3 Programming Environment **5**
3.1 Parallel Programming 5
3.2 The Tracing Tool 6

4 Basic Wrapper Structures **9**
4.1 Event Tracing 9
4.2 1-to-N Wrapper Structure 12
4.3 N-to-1 Wrapper Structure 14
4.4 N-to-N Wrapper Structure 15
4.5 N-to-1-to-N Wrapper Structure 17

5 The final Wrapper Implementations **21**
5.1 Message Passing Interface 21
5.2 Parallel Virtual Machine 30
5.3 Shared Memory 34

6 Testing the Wrappers **45**
6.1 Message Passing Interface 45
6.2 Parallel Virtual Machine 49
6.3 Shared Memory 51

7 Investigating the Collective Communication Functions **55**
7.1 Message Passing Interface 55
7.2 Parallel Virtual Machine 58
7.3 Shared Memory 59

8 Recommendations **61**

9 Conclusions **63**

Acknowledgement **65**

Appendix. **67**

References **91**

Index **93**

1 A brief Overview of the Company

As a member of the Hermann von Helmholtz-Gemeinschaft Deutscher Forschungszentren (HGF), the Forschungszentrum Jülich (FZJ) is a research institute with long-term research goals that have a great commitment and value to science and industry.

The Forschungszentrum Jülich was founded in 1956 as Kernforschungsanlage Jülich (KFA), a federal institute of the state Nordrhein Westfalen. In 1967 it became a limited liability company. The federal government of Germany gradually took over 90% of the financing, the government of Nordrhein Westfalen the remaining 10%. Nowadays about 4300 people perform their profession at the Forschungszentrum Jülich.

In the 42-year existence of the Forschungszentrum Jülich, the research priority has changed from nuclear energy science to an interdisciplinary field of study with the following main areas: structure of matter and materials research; information technology; energy technology; environmental precaution research and life sciences. In addition to interdisciplinarity, the second mainstay at the Forschungszentrum Jülich is the operation of large-scale research equipment, as a service for universities, other research institutions and industry.

The close links to the universities in Nordrhein Westfalen manifest in, e.g., the joint appointment of institute directors at the research centre to a chair at one of the universities in the federal state, the RWTH in Aachen. In addition to co-operations with universities and non-university research institutions, co-operation with the industry has continuously intensified. In 1995, the Forschungszentrum Jülich set up a record of 85 patent applications with the German Patent Office.

Zentralinstitut für Angewandte Mathematik

The Zentralinstitut für Angewandte Mathematik (ZAM) at the Forschungszentrum Jülich is responsible for the planning, realisation and operation of the central data processing and server computer systems including their communication systems. The research and developmental work of the Zentralinstitut für Angewandte Mathematik concentrate on calculations in scientific problems, the overlap between mathematics, informatics, technical and natural sciences. Applications processed on the central computer systems originate from several fields of study and vary from simulating chemical reactions to physical measurement analysis tools.

Since 1984 the Zentralinstitut für Angewandte Mathematik has started to deploy a number of supercomputers. Currently two massive parallel computer systems of type Cray T3E with 512 and 256 processors are available. Next to these, three vector supercomputers consisting of a Cray T90 with 16 processors and two Cray J90's with 16 and 12 processors are exploited. A massive parallel supercomputer of Intel is also available, an Intel Paragon XP/S with 151 processors. All these supercomputers are interconnected via several types of super fast computer networks. An IBM RS/6000 R50 with 8 processors, a large amount of external memory (about 40 Tb) and a number of terminals and workstations complete the whole.

2 Introduction

To enable an effective and efficient use of computer systems with more than one processor, the development of programming techniques and tools for these computer systems is essential. Networks of workstations and parallel computer systems are the most common types of computer systems with more than one processor. Especially the development of programming techniques and tools for massive parallel computer systems is a relatively unmatured science.

A parallel computer system contains a number of independent processors. If all processors have one common main memory, the computer system has a so-called shared memory structure. However, memory access conflicts in shared memory systems restrict the practical number of processors to 16 or 32. In a distributed memory system, all processors have their own main memory. A computer system based on a distributed memory structure can have a very large number of processors. Such a computer system is also called a massive parallel computer system. Because of cost aspects, more often standard processors are used in massive parallel computer systems. E.g., the Cray T3E uses DEC Alpha processors.

The more processors used in a parallel computer system, the more important becomes the interconnecting network between the processors. Due to the distributed memory structure of a parallel computer system, data needs to be exchanged between the processor main memories. This means that programming a computer system with distributed memory involves the programming of communication between the processors. Communication is generally implemented using functions of message passing libraries.

Currently, there are two commonly standardised and portable message passing libraries: Message Passing Interface (MPI) and Parallel Virtual Machine (PVM). In addition, vendors of parallel computers often provide their own optimised message passing library for their computer systems, e.g., Shared Memory (SHMEM) of Cray Research.

Due to the dynamic behaviour of the communicating processes in programs for parallel computers, understanding, debugging and tuning of such parallel programs is very complex. A method to analyse the behaviour of parallel programs is event tracing: during the execution of a parallel program, all important events (e.g., entering or leaving a routine, sending or receiving a message) are recorded with a time stamp and a process identifier in a special file called the event trace. After sorting this event trace according to the time stamps, it can be used to reconstruct the dynamic behaviour of the parallel program under investigation. Powerful event trace analysis tools are available to support the programmer in this task.

However, one problem remains. The parallel program has to be changed so that the necessary information about the events can be recorded. This process is called instrumentation. It should be done automatically because parallel programs can be large and complex. The Zentralinstitut für Angewandte Mathematik at the Forschungszentrum Jülich implements in association with Cray Research an instrumentation module for the Performance Analysis Tool (PAT) for the Cray T3E parallel computer systems. PAT instruments the executable program code. This has the advantage that PAT is independent from the programming language used. Instrumentation is done by inserting a call to special functions, called hooks, at the beginning and end of the routine that needs to be analysed.

Message passing libraries provide two categories of communication methods. Point-to-Point or 1-to-1 communication entails one sending and one receiving process. If more than two processes are involved in a single communication, a collective communication is considered. There are three typical forms of collective communication. A collective communication of type 1-to-N involves one sender and more than one receiver. A communication, for which a number of senders send a message to one receiver, is of type N-to-1. In the case that a communication resorts to more than one sender and more than one receiver, an N-to-M collective communication is considered.

Next to the usual Point-to-Point communication method, the SHMEM library provides so-called atomic communication functions. Atomic communication functions enable the manipulation of a remote processor's memory contents without informing the remote processor.

This report discusses the development, implementation and testing of the hooks for the collective communication functions of the message passing libraries available on the Cray T3E parallel computer systems. The implementation and testing of the hooks for the SHMEM atomic communication functions is also included.

Although compiling, debugging and other tools are available for popular scientific programming languages like Fortran, C and C++, the hooks for the collective communication functions of the message passing libraries and all other programs discussed in this report are written in the standard C programming language to increase portability.

Chapter Overview

In subsequent chapters, a processor of the Cray T3E parallel computer system is called a processing element. The next chapter gives an overview of the programming environment. After discussing some properties concerning parallel programs, it is explained how a performance analysis is accomplished. The fourth chapter is a discussion about general frameworks for the implementation of the hooks for collective communication functions. The frameworks result in a final implementation of the hooks in the fifth chapter, which are tested in the sixth chapter. The seventh chapter is a superficial investigation of how the collective communication functions of the message passing libraries are implemented internally. The eighth chapter gives some recommendations with regard to the results of chapter seven. Conclusions are given in the ninth chapter.

3 Programming Environment

Creating a tool that traces communication events between processing elements in a network of workstations or of a (massive) parallel computer involves parallel programming. After a description of how to develop a parallel program for the Cray T3E parallel computer system, a brief overview of the tracing tool is given.

3.1 Parallel Programming

A parallel program for the Cray T3E parallel computer systems runs under the UNIX-derived operating system UNICOS/mk. Beyond the POSIX-standard, this operating system includes important extensions like: multitasking to simultaneously use more than one processing element for a program, automatic restart of programs after a system interruption, multi-level memory hierarchy for UNIX file-systems and queuing of batch jobs. The message passing libraries MPI, PVM and SHMEM are included in the operating system as the Message Passing Toolkit (MPT).

A parallel program consists of only one source code. All processing elements on which the program runs execute the same code. During the execution of a parallel program, every processing element has its unique processing element number. In the case that a part of the code has to be executed by a subset of processing elements, the **`if`**-statement can be used to select the subset of processing elements by selecting their processing element numbers.

At the time of executing a parallel program on a number of processing elements, the execution processes are not synchronised. This means that two different processing elements might not execute an instruction of the parallel program exactly at the same time. The message passing libraries, however, enable the use of several synchronising functions.

It is especially important to consider that all processing elements execute the same code, but possibly not exactly at the same time, if input and output operations are considered. E.g., if one output file is used in a parallel program by all processing elements, the last processing element that changes the output file will overwrite all changes previously made by other processing elements.

To enable the use of functions of a message passing library in a parallel program, a header file needs to be included. Table 3.1 depicts the file names.

Library	C/C++	Fortran
MPI	‘mpi.h’	‘mpif.h’
PVM	‘pvm3.h’	‘fpvm3.h’
SHMEM	‘mpp/shmem.h’	‘mpp/shmem.fh’

Table 3.1. Header files for the message passing libraries.

The MPI_Init function has to be called once prior to using any function of the MPI library. If no communication is needed anymore in a parallel program that uses the MPI library, the MPI_Finalize function has to be called. In a parallel program that uses functions of the PVM library, the pvm_exit function has to be used to finalise the PVM message passing session.

The use of the MPI and PVM functions is explained in detail in [Dongarra] and [Beguelin1] respectively. For the SHMEM functions the reader is referred to [`man shmem`].

After compiling a parallel program, the `mpprun` command can be used to run the parallel program. The number '*`foo`*' in the option `–n` *`foo`* for this command indicates the number of processing elements that is used for the execution of the program. The command:

```
mpprun –n foo bar
```

executes the executable program code with filename '*`bar`*' using '*`foo`*' processing elements. During execution of the parallel program '*`bar`*', the processing element numbers range from 0 through '*`foo`*' - 1. For other options of the `mpprun` command, see [`man mpprun`].

3.2 The Tracing Tool

This section gives a brief overview of how an application can be analysed with PAT. For more detailed and up-to-date information about all features, see the documentation of PAT and [`man pat`]. The discussion includes visualisation.

A set of entrance and exit hooks for a routine is called a wrapper for that routine. Normally, default wrappers, which trace only the entrance and exit of a routine, are used. The wrappers for the communication functions of the message passing libraries also record how communication between the processing elements took place.

For any analysis of a parallel program for which collective communication functions of the message passing libraries are traced with PAT, the environment variable PAT_TRACE_COLLECTIVE has to be set to a space or 1. Table 3.2 depicts how to set this environment variable for some command shells.

Command shell	Example
makefile	`PAT_TRACE_COLLECTIVE=1`
sh, ksh	`export PAT_TRACE_COLLECTIVE=1`
csh, tcsh	`setenv PAT_TRACE_COLLECTIVE 1`

Table 3.2. Setting the environment variable PAT_TRACE_COLLECTIVE.

To accomplish an analysis with PAT the object modules of the application have to be re-linked with the PAT library and the necessary wrappers. The following files need to be linked to an application: 'libpat.a', 'pat.cld' and 'libwrapper.a'. If only functions of a specific message passing library are analysed, the file containing the library's wrappers: 'mpi_wrapper.o', pvm_wrapper.o' or 'sma_wrapper.o' (for the MPI, PVM and SHMEM library respectively), could be linked to the parallel program instead of the file 'libwrapper.a'.

All routines that become traced have to be listed in a file (e.g., '*`list`*'), one routine name per line. It is very important that the name of a routine is spelled exactly the same as used in the object code of the application. This includes the use of upper- and lower-case. Any routine listed in the '*`list`*' file is traced by default. However, after including the header file 'event_trace.h', it is possible to switch on and off tracing for specific parts of the application by using the functions EVENT_TRACE_ON and EVENT_TRACE_OFF.

To ease tracing of the functions of the message passing libraries, predefined '*list*' files are available. Their filenames include the string 'HOOK_LIST' and list all functions for which special wrappers are available.

Some remarks: In order to trace any function of the MPI library, the MPI_Init function has to be included in the '*list*' file. For the PVM library, any function starting with 'pvm_upk' has to be included in the '*list*' file if any of the functions: pvm_recv, pvm_nrecv, pvm_trecv, pvm_bcast or pvm_mcast have to be traced. Many functions of the SHMEM library are aliases for another function of the SHMEM library. It is important not to use the alias names but the name of the matching function in the 'HOOK_LIST' files.

An extra note applies to the analysis of parallel programs written in the programming language Fortran. Except for the MPI_SEND and MPI_RECV functions, MPI and PVM functions in a Fortran program call the corresponding C versions. So, because only wrappers were written for the C versions of these functions, the corresponding C function names must be used in the '*list*' file when tracing MPI or PVM functions. For SHMEM collective communication functions only, a similar situation appears. However, in this case the names of the C versions might be rather different. See section 5.3 and the on-line documentation of the concerning SHMEM collective communication function for the name of its C version and the 'HOOK_LIST' files to compose the '*list*' file.

The preparations above enable the final instrumentation of an application. This is done by the command:

```
pat –L –B list bar
```

where '*list*' is the name of the file containing the list of routines to be traced and '*bar*' the filename of the executable program code after re-linking with the PAT library and wrappers. The instrumentation process generates a file with suffix '.out' (e.g., *a.out*).

After running the instrumented application, a pif file containing the event trace is generated. The pif file is named like the file generated by the instrumentation process but has an extra suffix '.pif' (e.g., *a.out.pif*). The events in this file are already merged and sorted by PAT.

Visualisation

The small program 'piftest' enables a text-based overview of an event trace. The program VAMPIR by Pallas enables graphical visualisation of a recorded trace. However, first the trace format of the file with suffix '.pif' has to be converted to a VAMPIR trace file that has suffix '.bpv'. This is done with the `pif2bpv` command:

```
pif2bpv a.out.pif example.bpv
```

where '*a.out.pif*' is the generated pif file and '*example.bpv*' the file that can be used by VAMPIR to give a graphical visualisation of the recorded event trace.

4 Basic Wrapper Structures

After the introduction of some environmental properties, this chapter discusses general frameworks for the wrappers of the collective communication functions of the message passing libraries. The discussion is subdivided according to the different forms of collective communication. The next chapter uses the basic wrapper structures to develop the final wrapper implementations.

4.1 Event Tracing

In order to enable tracing of a routine call with a special wrapper, an entrance and exit hook has to be available for that routine. These hooks are special functions with the same arguments as the original routine and have the same name as the original routine but with an extra suffix: '_trace_entry' for the entrance hook and '_trace_exit' for the exit hook.

Because it is possible to disable tracing of a routine call at any time (See section 3.2), it first has to be checked whether the concerning routine must be traced or not. Investigating the (global) variable EVENT_TRACE_GATE does this. If the routine has to be traced, the value of EVENT_TRACE_GATE is 1. For any routine in the '`list`' file, called within this routine that does not need to be traced, the value of EVENT_TRACE_GATE has to be reset to 0 in the wrapper.

Each wrapper records entrance and exit of a routine call. In the entrance hook of a wrapper, the function EVENT_TRACE_SHORT_ENTRY(PAT_*name*) is used to record the entrance of the routine '*name*'. The function EVENT_TRACE_SHORT_EXIT(PAT_*name*) is used in the exit hook to record the exit of '*name*'.

As discussed in section 3.2, the environment variable PAT_TRACE_COLLECTIVE has to be set to enable tracing of collective communication functions. The (global) variable PAT_collective stores the status of the environment variable PAT_TRACE_COLLECTIVE for the wrappers of the functions of the message passing libraries. Its value is 1 if collective communication functions have to be traced.

The MPI library enables the use of so-called communicators. A communicator consists of a specified set of processing elements participating in a (number of) communication event(s). Every communicator that contains processing elements, of which any processing element is also in another communicator, has a unique number. However, any non-intersecting sets of processing elements specified as communicators have the same communicator number. The currently not yet implemented function PAT_comm(MPI_Comm *comm*), defined for the MPI wrappers only, will calculate a unique number for every set of processing elements specified as a communicator.

Next to communicators, the MPI and PVM libraries use tags to coordinate communication. However, MPI collective communication functions do not use tags. The values of the tag and communicator variables are also recorded during event tracing. See [Gropp] and [Dongarra] for more information about the use of communicators and tags.

Because not all functions of the message passing libraries use communicators and tags, dummy values are used while recording a communication event. The dummy values are defined in the file ‘wrapper.h’ that must be included for the wrappers of the message passing library. The file ‘wrapper.h’, depicted in figure 4.1, indicates, e.g., that the value of the dummy communicator is -1.

Figure 4.1. The file ‘wrapper.h’.

```
#ifndef __WRAPPER_H_
#define __WRAPPER_H_

/* -- dummy tag and communicator -- */
#define PAT_NOTAG  -1
#define PAT_NOCOMM -1

/* -- dummy tags used for MPI/SHMEM Collective Communication Routines -- */
#define PAT_1_TO_N_TAG   -1234
#define PAT_N_TO_1_TAG   -1235
#define PAT_N_TO_N_TAG   -1236

/* -- Compile-time flags for how to display collective N->N results -- */

#undef ALL_N_TO_N
#undef ALL_N_TO_1_TO_N

/* -- To get equal time stamps for collective functions -- */

#define PAT_settime(t) _tr_buf[_tr_next - EVENT_TRACE_WPTE + 2] = t
#define PAT_gettimeloc() _tr_buf + _tr_next + 2

#endif
```

The wrappers for the collective communication functions of the message passing libraries record how communication between the participating processing elements took place. In the communication event tracing function EVENT_TRACE(PAT_*name*, PAT_TRACE_*type*, *tag*, *pe*, *size*, *comm*), ‘*name*’ is the name of the routine for which the wrapper is. In the case that a sending event is recorded, ‘*type*’ is replaced by SEND. Is a receiving event concerned, ‘*type*’ is replaced by RECV. The variable of ‘*tag*’ records the tag value belonging to the communication event. The total number of bytes concerning the communication event is recorded by ‘*size*’. The communicator value is traced via the value of ‘*comm*’.

If the processing element that calls the EVENT_TRACE function belongs to a sending (receiving) part of a communication, the value of ‘*pe*’ denotes the processing element number of the receiving (sending) processing element. It is important that for ‘*pe*’ the correct processing element number is used. ‘*pe*’ must be the number that the processing element has in the total set of all processing elements drawn on by the `mpprun` command.

All communication events, recorded with the EVENT_TRACE function, are stored in a so-called event trace buffer. All parameters of the EVENT_TRACE function are stored in this buffer, including the processing element number of the processing element that called the function and the time at which the function was called. Finally, PAT copies the event trace buffer, after merging and sorting the recorded events, into a pif file containing the final event trace. Because all events are recorded with a time stamp, it is possible to reconstruct the dynamic behaviour of the parallel program.

Logical Patterns

The wrappers for the collective communication functions of the message passing libraries record logical patterns. This is done because it is not known how the collective communication functions implement collective communication internally. The logical patterns do not represent the real communication paths that occur according to the internal implementations of the collective communication functions. To obtain logical patterns while tracing the collective communication functions, the environment variable PAT_TRACE_COLLECTIVE has to be set. Chapter 7 discusses how the real communication paths of the collective communication functions could be obtained.

For some collective communication functions of type N-to-M it is not clear in what logical order communication takes place. Therefore, compile-time flags are introduced for how to record a collective N-to-M communication: as a star (all-to-1-to-all or N-to-1-to-N) or as a non-star (true all-to-all or N-to-N) pattern. The most realistic order is used as default for these functions. The compile-time flags are named: ALL_N_TO_1_TO_N and ALL_N_TO_N respectively. Note that collective communication functions of the MPI, PVM and SHMEM message passing libraries of type N-to-M always involve the same number of sending and receiving processing elements.

The logical patterns that are recorded for the collective communication functions are depicted in figure 4.2. Any logical pattern for the collective communication functions does not include communication paths between a single processing element. Only communication paths between different processing elements are recorded.

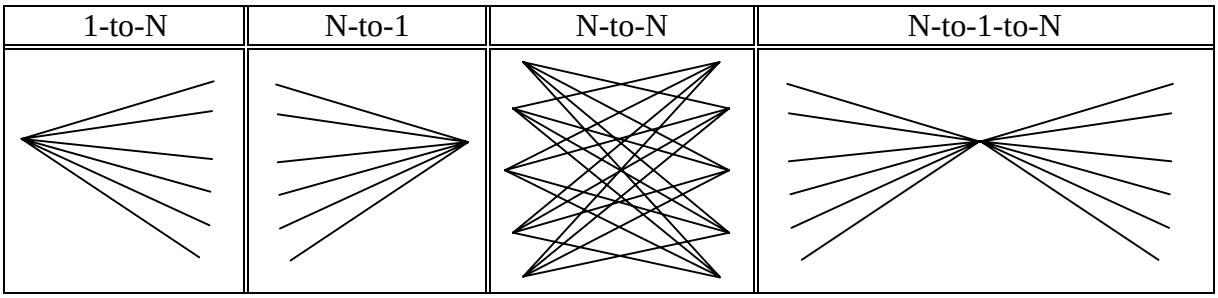

Figure 4.2. Logical patterns for the collective communication functions.

Time Manipulation

Generally, a loop construction is used for tracing a collective communication function. However, calling the EVENT_TRACE function in a loop results in a non-simultaneous start or finish of a communication process. Because logical patterns involve simultaneous starts and finishes of communication processes, time manipulation is necessary.

The functions PAT_settime(*t*) and PAT_gettimeloc(), defined in 'wrapper.h', enable time manipulation. PAT_gettimeloc() returns a pointer to the position at which the next time stamp will be stored by the EVENT_TRACE function. PAT_settime(*t*) resets the time stamp of the previous call of the function EVENT_TRACE to the value at which '*t*' points.

To enable the construction of the N-to-1-to-N logical pattern, a global variable has to be introduced, which stores the time at which the entrance hook is executed. It is named: starttime. The current time is returned by the function _rtc(), a UNICOS/mk system function.

4.2 1-to-N Wrapper Structure

In figure 4.3, which shows the general framework for the wrappers of 1-to-N collective communication functions, '*name*' has to be replaced by the name of the function for which the wrapper is. Note that no arguments of the function '*name*' are specified.

Figure 4.3. Basic wrapper structure for 1-to-N collective communication functions.

```
name_trace_entry(…) {

  int ii;
  int *timeptr;

  if (EVENT_TRACE_GATE) {
    EVENT_TRACE_GATE = 0;
    EVENT_TRACE_SHORT_ENTRY(PAT_name);
    if (PAT_collective) {
      if (my_pe_no == global_root) {
        timeptr = PAT_gettimeloc();              /* Remember location of
                                                    first time stamp  */
        for (ii = 0; ii < group_size; ii++) {
          if ( (ii != local_root) && (ii_size != 0) ) {
            EVENT_TRACE(PAT_name, PAT_TRACE_SEND,
                        tag, ii_global_pe, ii_size, comm);
            PAT_settime(*timeptr);               /* Reset time stamp  */
          }
        }
      }
    }
    EVENT_TRACE_GATE = 1;
  }
}

name_trace_exit(…) {
  if (EVENT_TRACE_GATE) {
    EVENT_TRACE_GATE = 0;
    if (PAT_collective) {
      if ( (my_pe_no != global_root) && (my_size != 0) ) {
        EVENT_TRACE(PAT_name, PAT_TRACE_RECV,
                    tag, global_root, my_size, comm);
      }
    }
    EVENT_TRACE_SHORT_EXIT(PAT_name);
    EVENT_TRACE_GATE = 1;
  }
}
```

Generally, functions of the concerning message passing library are used to determine the italic variables (except for '*name*') out of the function arguments. The (global) variable EVENT_TRACE_GATE is manipulated to prevent tracing of internally called message passing functions that could be in the '*list*' file, see section 4.1.

After recording the entrance of the function '*name*' by using the function EVENT_TRACE_SHORT_ENTRY(PAT_*name*) in the entrance hook, it has to be checked whether the (global) variable PAT_collective is set. If so, N - 1 send events have to be recorded by the sending processing element. This processing element is called the root processing element.

Because all processing elements that participate in a call of '*name*' execute the entrance hook of '*name*' if the function '*name*' is in the '`list`' file, the **`if`**-statement is used to select the root processing element. Other processing elements than the root processing element do not have to record any events while executing the entrance hook. The processing element number of a processing element is available via its value of '*my_pe_no*', which is the number of the processing element in the total set of all processing elements drawn on by the `mpprun` command. The value of '*global_root*' is determined via the arguments of the function '*name*'. Note that '*global_root*' must represent the number of the root processing element in the total set of all processing elements drawn on by the `mpprun` command.

After saving the location of the time stamp of the event to be recorded first, the root processing element executes a **`for`**-loop to record the N - 1 send events. Recording an event is done in the **`for`**-loop by using the EVENT_TRACE function and setting the time stamp of the event equal to that of the send event which is recorded first.

The variable '*group_size*' is determined via the arguments of the function '*name*' and indicates the number of processing elements participating in the call of '*name*', including the root processing element. Because the logical pattern for any collective communication function consists of communication paths between unequal processing elements only, the root processing element must be excluded from the **`for`**-loop to prevent recording a send event to itself. Note that ii ranges from 0 through '*group_size*' - 1, which is generally unequal to the total number of processing elements minus 1 drawn on by the `mpprun` command. This means that '*local_root*' is the rank of the root processing element in the participating set of processing elements that called '*name*'. The root processing element only records a send event to all those (non-root) participating processing elements to which a non-zero number of bytes, available via '*ii_size*', is sent. The value of '*ii_size*' is determined via the arguments of the function '*name*'.

The value of '*ii_global_pe*' is the processing element number of the processing element in the total set of processing elements drawn on by the `mpprun` command of which the rank in the set of participating processing elements equals ii.

In all wrappers for the MPI and SHMEM 1-to-N collective communication functions, the value of '*tag*' equals the value of the dummy tag PAT_1_TO_N_TAG. Because PVM 1-to-N collective communication functions use tags to coordinate communication, the value of '*tag*' in their wrappers is determined via the arguments of the function '*name*'. The value of '*comm*' in the wrappers of PVM and SHMEM 1-to-N collective communication functions is equal to the value of the dummy PAT_NOCOMM, whereas '*comm*' is substituted by PAT_comm(*comm*) in the wrappers for the MPI 1-to-N collective communication functions. The value of '*comm*' in PAT_comm(*comm*) is determined via the arguments of the MPI function.

If the function '*name* ' is in the '`list`' file, every processing element participating in a call of the function '*name*' executes the exit hook of '*name*'. All processing elements, unequal to the root processing element, which received a non-zero number of bytes (available via '*my_size*'), record a receive event in the exit hook. To record the exit of '*name*', the function EVENT_TRACE_SHORT_EXIT(PAT_*name*) is finally used.

4.3 N-to-1 Wrapper Structure

In the general framework for the wrappers of N-to-1 collective communication functions, given in figure 4.4, '*name*' has to be replaced by the name of the function for which the wrapper is. No arguments are specified for the function '*name*'.

Figure 4.4. Basic wrapper structure for N-to-1 collective communication functions.

```
name_trace_entry(…) {
  if (EVENT_TRACE_GATE) {
    EVENT_TRACE_GATE = 0;
    EVENT_TRACE_SHORT_ENTRY(PAT_name);
    if (PAT_collective) {
      if ( (my_pe_no != global_root) && (my_size != 0) ) {
        EVENT_TRACE(PAT_name, PAT_TRACE_SEND,
                    tag, global_root, my_size, comm);
      }
    }
    EVENT_TRACE_GATE = 1;
  }
}

name_trace_exit(…) {

  int ii;
  int *timeptr;

  if (EVENT_TRACE_GATE) {
    EVENT_TRACE_GATE = 0;
    if (PAT_collective) {
      if (my_pe_no == global_root) {
        timeptr = PAT_gettimeloc();              /* Remember location of
                                                    first time stamp  */
        for (ii = 0; ii < group_size; ii++) {
          if ( (ii != local_root) && (ii_size != 0) ) {
            EVENT_TRACE(PAT_name, PAT_TRACE_RECV,
                        tag, ii_global_pe, ii_size, comm);
            PAT_settime(*timeptr);               /* Reset time stamp  */
          }
        }
      }
    }
    EVENT_TRACE_SHORT_EXIT(PAT_name);
    EVENT_TRACE_GATE = 1;
  }
}
```

Because the general framework of figure 4.4 is basically a mirror image of figure 4.3, the variables of figure 4.4 have the same meaning and origin as in section 4.2.

If the function '*name*' must be traced, all processing elements other than the receiving root processing element, which send a non-zero number of bytes to the root processing element, are selected in the entrance hook for recording a send event. Therefore, the processing element number '*my_pe_no*' of a processing element is compared with the value of '*global_root*', representing the processing element number of the root processing element in the total set of processing elements drawn on by the `mpprun` command. The values of '*global_root*' and '*my_size*' are determined via the arguments of the function '*name*'.

For the wrappers of the MPI N-to-1 collective communication functions, the value of '*tag*' is equal to the value of PAT_N_TO_1_TAG, but for the PVM functions of this type, '*tag*' is determined via the arguments of the function '*name*'. The value of '*comm*' in the wrappers of PVM N-to-1 collective communication functions equals the value of the dummy PAT_NOCOMM, whereas '*comm*' is substituted by PAT_comm(*comm*) in the wrappers for the MPI N-to-1 collective communication functions. The value of '*comm*' in PAT_comm(*comm*) is determined via the arguments of the function '*name*'. Note that no N-to-1 collective communication functions are available in the SHMEM library.

After saving the location of the time stamp of the first event to be recorded, the receiving root processing element executes a **`for`**-loop to record N - 1 receive events in the exit hook. The local variable ii ranges from 0 to '*group_size*' - 1, which is determined via the arguments of the function '*name*', in order to record a receive event from the processing element of which the rank in the set of participating processing elements in the call of '*name*' equals ii. The value of '*ii_size*', representing the number of bytes received from the processing element with rank ii, is determined via the arguments of the function '*name*'. Because ii represents ranks, '*ii_global_pe*' must be the processing element number of the processing element with rank ii. After recording a receive event using the EVENT_TRACE function, the root processing element sets the time stamp of the recorded receive event to the value of the receive event recorded first.

4.4 N-to-N Wrapper Structure

The general framework for the wrappers of the N-to-N collective communication functions basically consists of a combination of the frameworks discussed in section 4.2 and 4.3. Every processing element has communication paths to all other processing element participating in the call of an N-to-N collective communication function. Therefore, no root processing element is defined for N-to-N collective communication functions. Figure 4.5, on the next page, depicts the general framework for the wrappers of the N-to-N collective communication functions.

If the N-to-N collective communication function '*name*' is in the '`list`' file, all participating processing elements in the call of '*name*' save the location of the first time stamp to be recorded after checking PAT_collective in the entrance hook. Because all participating processing elements act like a root processing element, all participating processing elements execute a **`for`**-loop to record N - 1 send events. The value of '*group_size*', which is determined via the arguments of the function '*name*', is equal to the number of participating processing elements in the call of '*name*'.

A processing element records a send event only to all those other participating processing elements to which a non-zero the number of bytes is sent. Therefore, the value of ii is compared with the value of '*my_rank*', which represents the rank of the processing element in the set of participating elements that executes the **`for`**-loop. The number of bytes, sent by the processing element with rank ii in the participating set of processing elements, is stored in '*ii_size*'. Both '*my_rank*' and '*ii_size*' are determined via the arguments of the function '*name*'.

The value of '*ii_global_pe*' in figure 4.5 represents the processing element number of the processing element with rank ii in the set of participating processing elements.

Figure 4.5. Basic wrapper structure for N-to-N collective communication functions.

```
name_trace_entry(…) {

  int ii;
  int *timeptr;

  if (EVENT_TRACE_GATE) {
    EVENT_TRACE_GATE = 0;
    EVENT_TRACE_SHORT_ENTRY(PAT_name);
    if (PAT_collective) {
      timeptr = PAT_gettimeloc();                /* Remember location of
                                                    first time stamp  */
      for (ii = 0; ii < group_size; ii++) {
        if ( (ii != my_rank) && (ii_size != 0) ) {
          EVENT_TRACE(PAT_name, PAT_TRACE_SEND,
                      PAT_N_TO_N_TAG, ii_global_pe, ii_size, comm);
          PAT_settime(*timeptr);                 /* Reset time stamp  */
        }
      }
    }
    EVENT_TRACE_GATE = 1;
  }
}

name_trace_exit(…);

  int ii;
  int *timeptr;

  if (EVENT_TRACE_GATE) {
    EVENT_TRACE_GATE = 0;
    if (PAT_collective) {
      timeptr = PAT_gettimeloc();                /* Remember location of
                                                    first time stamp  */
      for (ii = 0; ii < group_size; ii++) {
        if ( (ii != my_rank) && (ii_size != 0) ) {
          EVENT_TRACE(PAT_name, PAT_TRACE_RECV,
                      PAT_N_TO_N_TAG, ii_global_pe, ii_size, comm);
          PAT_settime(*timeptr);                 /* Reset time stamp  */
        }
      }
    }
    EVENT_TRACE_SHORT_EXIT(PAT_name);
    EVENT_TRACE_GATE = 1;
  }
}
```

Because no N-to-N collective communication functions are available in the PVM library, the value of the tag recorded by the wrappers for N-to-N collective communication functions is equal to the value of PAT_N_TO_N_TAG. For the wrappers of MPI N-to-N collective communication functions, '*comm*' is substituted by PAT_comm(*comm*), in which '*comm*' is determined via the arguments of '*name*'. Because SHMEM N-to-N collective communication functions make no use of communicators, '*comm*' equals the dummy PAT_NOCOMM in their wrappers.

After recording a send event, the time stamp is set equal to the time stamp of the first send event recorded. Note that, generally, the first time stamp is different for every participating processing element in the call of '*name*'.

Every processing element executing the exit hook for the N-to-N collective communication functions records receive events from all other participating processing element in the call of the function '*name*' from which a non-zero number of bytes is received. Accept for the type of events recorded and the call of the EVENT_TRACE_SHORT_ENTRY(PAT_*name*) and EVENT_TRACE_SHORT_EXIT(PAT_*name*) functions, the entrance and exit hooks for the N-to-N collective communication functions are equal in figure 4.5.

4.5 N-to-1-to-N Wrapper Structure

Principally, the N-to-1-to-N logical pattern consists of an N-to-1 logical pattern followed by a 1-to-N logical pattern. Because no root processing element is available, as is for the 1-to-N and N-to-1 collective communication functions, a root processing element has to be chosen to construct the N-to-1-to-N logical pattern. Figure 4.6 gives the general framework for the wrappers of collective communication functions with an N-to-1-to-N logical pattern.

If '*name*' is in the '`list`' file, the value of '*group_size*', determined via the arguments of the function '*name*', is equal to the number of processing elements that participate in the call of the entrance and exit hooks of the function '*name*'. Dividing '*group_size*' by 2 results in a rank number for a fictive root processing element in the set of participating processing elements, which generally ends up in a symmetrical star pattern recorded by the wrapper for the collective communication functions with an N-to-1-to-N logical pattern. In figure 4.6, '*global_pe*' has to be replaced by functions of the concerning message passing library to determine the processing element number of a processing element's rank given as argument to '*global_pe*'. The variable local_root (in the exit hook only) stores the rank and the variable global_root the processing element number of the fictive root processing element.

Next to choosing a fictive root processing element, time manipulation is needed to construct a symmetrical star pattern. The fictive root processing element should record receive and send events with a time stamp equal to the arithmetic mean of the times of entering and leaving the function '*name*'. The time at which the function '*name*' is entered can only be determined in the entrance hook and is saved in the global variable starttime. The time at which the function '*name*' is left can not be determined until executing the exit hook. This means that in the entrance hook only send events for the N-to-1 part can be recorded. All other events have to be recorded in the exit hook. Time manipulation is necessary to record the arithmetic mean time for the events recorded by the fictive root processing element.

After checking the variable PAT_collective, the value of the variable global_root is determined in the entrance hook. The number of bytes sent by a participating processing element, stored in '*my_send_size*', is determined via the arguments of the function '*name*'. The value of '*my_pe_no*' is equal to the number of the processing element in the total set of processing elements drawn on by the `mpprun` command. The **`if`**-statement selects all participating processing elements other that the fictive root processing element to record a send event. The remaining participating processing elements, including the fictive root processing element, however, save the current time in the global variable starttime.

In wrappers for the collective communication functions with an N-to-1-to-N logical pattern of the MPI library, '*comm*' has to be replaced by PAT_comm(*comm*), in which '*comm*' is determined via the arguments of the function '*name*'. Otherwise, '*comm*' is equal to the dummy PAT_NOCOMM.

Figure 4.6. Basic wrapper structure for N-to-1-to-N collective communication functions.

```
name_trace_entry(…) {

  int ii;
  int *timeptr;

  int global_root;

  if (EVENT_TRACE_GATE) {
    EVENT_TRACE_GATE = 0;
    EVENT_TRACE_SHORT_ENTRY(PAT_name);
    if (PAT_collective) {
                                          /* Determine value for root  */
      global_root = global_pe(group_size/2);
                                          /* N to 1 part: Send         */
      if ( (my_pe_no != global_root) && (my_send_size != 0) ) {
         EVENT_TRACE(PAT_name, PAT_TRACE_SEND,
                     PAT_N_TO_1_TAG, global_root, my_send_size, comm);
      }
      else {
        starttime = _rtc();               /* Determine root's starttime */
      }
    }
    EVENT_TRACE_GATE = 1;
  }
}

name_trace_exit(…) {

  int ii;
  int *timeptr, recvtime;

  int local_root, global_root;

  if (EVENT_TRACE_GATE) {
    EVENT_TRACE_GATE = 0;
    if (PAT_collective) {
      local_root = group_size/2;          /* Determine value for root  */
      global_root = global_pe(local_root);

      if (my_pe_no == global_root) {
        recvtime = (_rtc() - starttime)/2 + starttime;

        for (ii = 0; ii < group_size; ii++) {
                                          /* N to 1 part: Receive      */
          if ( (ii != local_root) && (ii_send_size != 0) ) {
            EVENT_TRACE(PAT_name, PAT_TRACE_RECV,
                        PAT_N_TO_1_TAG, ii_global_pe, ii_send_size, comm);
            PAT_settime(recvtime);
          }
        }

        for (ii = 0; ii < group_size; ii++) {
                                          /* 1 to N part: Send         */
          if ( (ii != local_root) && (ii_recv_size != 0) ) {
            EVENT_TRACE(PAT_name, PAT_TRACE_SEND,
                        PAT_1_TO_N_TAG, ii_global_pe, ii_recv_size, comm);
            PAT_settime(recvtime + 1);
          }
        }
      }
```

```
      else {
                                              /* 1 to N part: Receive     */
        if (my_recv_size != 0) {
          EVENT_TRACE(PAT_name, PAT_TRACE_RECV,
                      PAT_1_TO_N_TAG, global_root, my_recv_size, comm);
        }
      }

    }
    EVENT_TRACE_SHORT_EXIT(PAT_name);
    EVENT_TRACE_GATE = 1;
  }
}
```

Collective communication functions with an N-to-1-to-N logical pattern are not available in the PVM library. Therefore, the dummy tags PAT_N_TO_1_TAG and PAT_1_TO_N_TAG are respectively used for the N-to-1 and 1-to-N parts of the N-to-1-to-N logical pattern.

In the exit hook, the value of local_root is first calculated after checking PAT_collective. Because the value of global_root in the exit hook, calculated out of local_root, is equal to the value of global_root in the entrance hook, no extra global variable has to be introduced. The fictive root processing element is selected by the **`if`**-statement to record N - 1 receive events for the N-to-1 part and N - 1 send events for the N-to-1 part of the N-to-1-to-N logical pattern. All participating processing elements other than the fictive root processing element are selected for recording a single receive event.

The fictive root processing element first calculates the arithmetic mean of the times of entering and leaving the function '*name*' and saves the result in recvtime. In the first **`for`**-loop, receive events are recorded from all participating processing elements other than the fictive root processing element which sent a non-zero number of bytes. The number of bytes, sent by the participating processing element with rank ii, is stored in '*ii_send_size*', determined via the arguments of the function '*name*'. The time stamps of these receive events are set to the time stored in recvtime.

After finishing the N-to-1 part of the N-to-1-to-N logical pattern, the fictive root processing elements executes a second **`for`**-loop to record a send event to all participating processing elements other than the fictive root processing element which received a non-zero number of bytes. The number of bytes, received by the participating processing element with rank ii, is stored in '*ii_recv_size*', which is determined via the arguments of the function '*name*'. As discussed in section 3.2, PAT finally merges and sorts the recorded events. However, the event merging and sorting algorithm of PAT is not able to handle send and receive events with a same time stamp. To prevent getting a corrupted event trace, the recorded time stamps for the send events in the 1-to-N part are set to the time stored in recvtime plus 1 (the minimal time interval).

Of all participating processing elements other than the fictive root processing element, selected in the **`else`**-part of the main **`if`**-statement, it is first checked whether the number of bytes received is non-zero. If so, the processing elements all record a single receive event from the fictive root processing element, finishing the 1-to-N part of the N-to-1-to-N logical pattern.

5 The final Wrapper Implementations

This chapter discusses the final implementation of the wrappers for the collective communication functions of the message passing libraries. The discussion includes the implementation of the wrappers for the atomic communication functions of the SHMEM library. Main subject is the calculation of the number of bytes recorded with an event. Note that the final wrapper implementations are copyrighted by the Zentralinstitut für Angewandte Mathematik and can therefore not be published in this report.

5.1 Message Passing Interface

Efficiently implementing the wrappers for the MPI collective communication functions requires only minor adaptation of the general frameworks discussed in chapter 4. Note that the value of PAT_collective is only available if the function MPI_Init is instrumented. In the wrapper for the MPI_Init function, the value of PAT_collective is initialised according to the value of the environment variable PAT_TRACE_COLLECTIVE.

Because communication functions of the MPI library use communicators to coordinate a communication, an argument, named '*comm*' of type MPI_Comm, is available for all MPI communication functions. The communicator type MPI_Comm is defined in the MPI library. To determine the size of the communicator '*comm*', represented by '*group_size*', the function MPI_Comm_size(MPI_Comm *comm*, **`int`** **group_size*) is used. The value of '*group_size*' equals the number of processing elements in the communicator '*comm*'. The rank of a processing element in the communicator '*comm*' can be determined by using the function MPI_Comm_rank(MPI_Comm *comm*, **`int`** **rank*).

The MPI library also defines the type MPI_Datatype to declare the type of the data concerning a communication. The function MPI_Type_size(MPI_Datatype *datatype*, **`int`** **size*) calculates the size in bytes of '*datatype*', declared as MPI_Datatype.

In the wrappers for the MPI collective communication functions, the value PAT_my_pe, defined in the library of PAT, equals the processing element number in the total set of all processing elements drawn on by the `mpprun` command. The arguments of 1-to-N and N-to-1 collective communication functions include the rank of the root processing element in the set of participating processing elements, stored in the integer '*root*'. The MPI function rank_to_pe(**`int`** *rank*, MPI_Comm *comm*) is used to calculate the processing element number of a processing element if its rank in the set of participating processing elements, defined by the communicator '*comm*', is known.

1-to-N Collective Communication Functions

The MPI library enables the use of three 1-to-N collective communication functions: MPI_Bcast, MPI_Scatter and MPI_Scatterv. The MPI_Bcast function broadcasts data elements of a root processing element to all other participating processing elements in the call of the MPI_Bcast function. The MPI_Scatter and MPI_Scatterv functions are used to scatter elements of a data array from the root processing element to the other participating processing elements.

☞ MPI_Bcast

The MPI_Bcast function has the following arguments:

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype,
              int root, MPI_Comm comm)
```

The pointer '*buffer*' is used by the root processing element to indicate the start address of the data elements that have to be sent. The receiving processing elements start storing the received data elements at the addresses pointed to by '*buffer*'. The number of data elements that the root processing element sends to every other participating processing element is stored in '*count*'. This is equal to the number of data elements that every participating processing element other than the root processing element receives. The type of the data elements is indicated by '*datatype*'.

Because the same amount of data elements is sent to all participating processing elements, it is checked whether '*count*' is non-zero at selecting the root processing element to execute the **for**-loop. Due to the implementation of the **&&**-operation in C, only the selected root processing element checks whether '*count*' is non-zero. Doing this increases the execution speed of the **for**-loop executed by the root processing element to record N - 1 send events in the entrance hook. Note that the processing element number of the root processing element has to be calculated by all participating processing elements before the **if**-statement can be used to select the root processing element.

The number of bytes recorded for every send and receive event is equal to the number of data elements sent by the root processing element multiplied by the number of bytes needed to store a data element of type '*datatype*'.

☞ MPI_Scatter

The MPI_Scatter function uses the following arguments:

```
int MPI_Scatter(void *sendbuf, int scount, MPI_Datatype stype,
                void *recvbuf, int rcount, MPI_Datatype rtype,
                int root, MPI_Comm comm)
```

According to increasing rank numbers in the set of participating processing elements, the root processing element evenly scatters data elements of type '*stype*' in the array '*sendbuf*' to all participating processing elements. The number of data elements that the root processing element sends to every participating processing element is given by '*scount*'. Therefore, the number of bytes recorded for a send event of the root processing element equals the multiplication of '*scount*' with the size in bytes of a data element of type '*stype*'.

The data is received in the array '*recvbuf*'. '*rcount*' indicates the number of data elements received by every participating processing element and should be equal to '*scount*'. The type of the received data elements is '*rtype*', which should be equal to '*stype*'. Although not recorded by the 1-to-N logical pattern, the '*recvbuf*' array of the root processing element is also changed. The number of bytes recorded for a receive event is equal to '*rcount*' multiplied by the number of bytes needed to store a data element of type '*rtype*'.

The number of data elements in '*sendbuf*' must at least be equal to the number of participating processing elements (including the root processing element) multiplied by '*scount*'. The number of data elements in '*recvbuf*' must be at least '*rcount*'.

To minimise execution time of the **`for`**-loop it is checked whether '*scount*' is non-zero at the time of selecting the root processing element for executing the **`for`**-loop in the entrance hook to record N - 1 send events. This is possible because the same number of data elements is sent to all participating processing elements.

☞ MPI_Scatterv

A more universal scatter than the MPI_Scatter function is the MPI_Scatterv function:

`int` MPI_Scatterv(**`void`** **sendbuf*, **`int`** **scounts*, **`int`** **displ*, MPI_Datatype *stype*,
`void` **recvb*, **`int`** *rcount*, MPI_Datatype *rtype*,
`int` *root*, MPI_Comm *comm*)

The MPI_Scatterv function enables scattering a different number of data elements to the different participating processing elements. The order in which the root processing element sends data elements to the participating processing elements is the order of increasing ranks of the participating processing elements. The root processing element stores the data elements of type '*stype*' that have to be sent in the array '*sendbuf*'. The number of data elements that has to be sent to the participating processing element with rank ii is stored in '*scounts*[ii]'. The array '*displ*' gives the positions of the data elements in '*sendbuf*' from which the root processing element starts sending to a specific participating processing element.

A send event recorded by the root processing element in the entrance hook is recorded with the number of bytes equal to the multiplication of the number of data elements sent to a participating processing element with the number of bytes needed to store a data element of type '*stype*'.

The received data elements of type '*rtype*', which should be equal to '*stype*', are stored in the array '*recvb*'. The number of received data elements equals the value of '*rcount*'. The value of '*rcount*' might be different for every participating processing element. In the exit hook, a receive event is recorded with the multiplication of '*rcount*' with the number of bytes needed to store a data element of type '*rtype*' for the number of bytes received. Note that, although not recorded by the 1-to-N logical pattern, the '*recvb*' array of the root processing element is also changed.

N-to-1 Collective Communication Functions

Three N-to-1 collective communication functions are available in the MPI library: MPI_Gather, MPI_Gatherv and MPI_Reduce. The MPI_Gather and MPI_Gatherv functions are the mirror functions of MPI_Scatter and MPI_Scatterv respectively. The MPI_Reduce function is a gather function that enables the use of a specific operation applicable on the data gathered from the different participating processing elements. The operation has to be declared as MPI_Op. The operation type MPI_Op is defined in the MPI library.

☞ MPI_Gather

The MPI_Gather function must be called with the following arguments:

int MPI_Gather(**void** **sendbuf*, **int** *scount*, MPI_Datatype *stype*,
void **recvb*, **int** *rcount*, MPI_Datatype *rtype*,
int *root*, MPI_Comm *comm*)

Every participating processing element in the call of the MPI_Gather function stores '*scount*' data elements of type '*stype*' in the send buffer indicated by '*sendbuf*'. The number of bytes that is sent to the root processing element is equal to the multiplication of '*scount*' with the number of bytes needed to store a data element of type '*stype*'.

The root processing element stores the received data elements of type '*rtype*', which should be equal to '*stype*', in the array named '*recvb*'. The number of data elements received from every participating processing elements is indicated by '*rcount*'. Therefore, the length of the array '*recvb*' must at least be '*rcount*' multiplied by the number of participating processing elements, including the root processing element. Note that also data elements of the array '*sendbuf*' of the root processing element are copied into '*recvb*'. This is not recorded in the logical pattern of the N-to-1 collective communication function.

☞ MPI_Gatherv

The arguments for the MPI_Gatherv function are:

int MPI_Gatherv(**void** **sendbuf*, **int** *scount*, MPI_Datatype *stype*, **void** **recvb*,
int **rcounts*, **int** **disps*, MPI_Datatype *rtype*,
int *root*, MPI_Comm *comm*)

The array '*sendbuf*' stores '*scount*' data elements of type '*stype*' in every participating processing element. These data elements are sent to the root processing element. Therefore, the number of bytes recorded with a send event is equal to the multiplication of '*scount*' with the size in bytes of a data element of type '*stype*'. Note that '*scount*' might be different for the different participating processing elements.

The root processing element receives the data elements in the array '*recvb*'. The type of these received data elements is '*rtype*' and should be equal to '*stype*'. The number of data elements received from the processing element with rank ii is stored in '*rcounts*[ii]'. The array '*disps*' contains the positions in '*sendbuf*' from which the root processing element starts receiving from a specific participating processing element. Although not recorded by the N-to-1 logical pattern, the root processing element's data elements in '*sendbuf*' are copied into '*recvb*' on the correct positions.

☞ MPI_Reduce

For the MPI_Reduce functions, the following arguments must be specified:

int MPI_Reduce(**void** **sendb*, **void** **recvb*, **int** *count*, MPI_Datatype *dt*,
MPI_Op *op*, **int** *root*, MPI_Comm *comm*)

Every participating processing element stores '*count*' data elements in a send buffer called '*sendb*'. The root processing element also has available a receiving array '*recvb*' for '*count*' data elements. The i^{th} data element of every '*sendb*' array of the participating elements contributes to the result calculated with the associative function '*op*', which is stored at the i^{th} position of the receiving buffer '*recvb*'. The function '*op*' must be able to handle data of type '*dt*'. Note that '*recvb*' must have a length equal to '*count*' and that the '*sendb*' array of the root processing element contributes to the results calculated with the '*op*' function.

The number of bytes recorded for the send and receive events is equal to the multiplication of '*count*' with the number of bytes needed to store a data element of type '*dt*'. In the exit hook, it is checked whether '*count*' is non-zero at selecting the root processing element for executing the **`for`**-loop to record N - 1 receive events.

N-to-M Collective Communication Functions with N-to-N (default) Pattern

In the MPI library, seven N-to-M collective communication functions are available of which for two the N-to-N logical pattern is recorded by the concerning wrapper only: MPI_Alltoallv and MPI_Scan. The wrappers for the functions MPI_Allgatherv and MPI_Alltoall record the N-to-N logical pattern as default and are discussed first.

☞ MPI_Allgatherv

The MPI_Allgatherv function, which is a more universal version of the MPI_Allgather function, uses the following arguments:

```
int MPI_Allgatherv(void *sendbuf, int scount, MPI_Datatype stype,
                   void *recvb, int *rcounts, int *disps,
                   MPI_Datatype rtype, MPI_Comm comm)
```

The effect of the MPI_Allgatherv function is similar to executing a MPI_Gatherv function by all participating processing elements successively being root processing element.

All participating processing elements store '*scount*' data elements of type '*stype*' in the array '*sendbuf*', which are sent to all other participating processing elements. The value of '*scount*' might be different for the different participating processing elements. In the entrance hook for recording the default N-to-N logical pattern, '*scount*' is checked to be non-zero when the global variable PAT_collective is checked. If PAT_collective is unequal to 1, no participating processing element records send events. However, if PAT_collective equals 1, only the participating processing elements for which '*scount*' is non-zero record send events. The number of bytes recorded by a participating processing element for the send events equals the multiplication of its value for '*scount*' with the size in bytes of a data element of type '*stype*'.

In the exit hook for recording the default N-to-N logical pattern, '*rcounts*[ii]' data elements are received from the processing element with rank ii in the set of participating processing elements. All the received data elements, which are of type '*rtype*' (should be equal to '*stype*'), are stored in the array '*recvb*'. The value of '*disps*[ii]' for the participating processing element with rank ii gives the start position from which its received data elements are stored in '*recvb*'. The number of bytes recorded for a receive event from the participating processing element with rank ii is equal to the value of '*rcounts*[ii]' multiplied by the number of bytes needed to store a data element of type '*rtype*'.

After executing the MPI_Allgatherv function, the values in '*recvb*' are the same for all participating processing elements. Defining the compile-time flag ALL_N_TO_1_TO_N enables recording the alternative N-to-1-to-N logical pattern. The N-to-N logical pattern is recorded by default because the number of bytes that is sent to the participating processing elements may influence the internal implementation of the MPI_Allgatherv function.

If the N-to-1-to-N logical pattern has to be recorded, recording the N-to-1 part is similar to the logical pattern recorded for the MPI_Gatherv function. The N-to-1 part of the N-to-1-to-N logical pattern is explained as: The fictive root processing element receives from all other participating processing elements their data elements and stores them in his '*recvb*'. In the 1-to-N part, the fictive root processing element must broadcast the array '*recvb*' to all other participating processing elements. However, it is not known directly by the arguments of MPI_Allgatherv how large the total number of received data elements in '*recvb*' is. Therefore, the variable total is introduced. A **`for`**-loop calculates the total number of received data elements in '*recvb*' by summing all values of '*rcounts*[ii]' for all ranks ii. The events recorded in the 1-to-N part of the N-to-1-to-N logical pattern record for the number of bytes the multiplication of total with the size in bytes of a data element of type '*rtype*'.

Note that the variable total must be checked for being non-zero both by the fictive root processing element and all other participating processing elements because the value of total is unavailable from the arguments of MPI_Allgatherv. If total equals zero, no communication paths have to be recorded at all because total is only zero if the value of '*rcounts*[ii]' is zero for all ii.

☞ MPI_Alltoall

The MPI_Alltoall function is called with the following arguments:

`int` MPI_Alltoall(**`void`** **sendbuf*, **`int`** *scount*, MPI_Datatype *stype*, **`void`** **recvb*,
`int` *rcount*, MPI_Datatype *rtype*, MPI_Comm *comm*)

Every participating processing element in a call of the MPI_Alltoall function evenly scatters the data elements of type '*stype*' in the array '*sendbuf*' to all other participating processing elements. The number of data elements that is sent to all other participating processing elements is represented by '*scount*'. In the entrance hook of the N-to-N logical pattern, which is recorded by default, the number of bytes recorded for a send event equals the multiplication of '*scount*' with the size in bytes of a data element of type '*stype*'.

The received data is stored in the array '*recvb*'. The type of the data elements in this array is '*rtype*', which should be equal to '*stype*'. '*rcount*' indicates the number of data elements received from every other participating processing element and should be equal to '*scount*'. The number of bytes recorded for a receive event is equal to the multiplication of '*rcount*' with the number of bytes needed to store a data element of type '*rtype*'.

Because the array '*recvb*' of all participating processing elements contain the same data after executing the MPI_Alltoall function, defining the compile-time flag ALL_N_TO_1_TO_N enables recording an N-to-1-to-N logical pattern. Generally, a faster algorithm for the internal implementation of the MPI_Alltoall function would result in an N-to-N communication pattern. Because this implementation might not be very complex, the N-to-N logical pattern is recorded by default.

Recording the N-to-1 part of an N-to-1-to-N logical pattern for the MPI_Alltoall function is similar to recording the N-to-1 logical pattern for the MPI_Scatter function. The 1-to-N part of the N-to-1-to-N logical pattern represents the broadcast of '*group_size*' arrays '*recvb*' that are received by the fictive root processing element from all other participating processing elements in the N-to 1 part. The variable '*group_size*' represents the number of all participating processing elements, including the fictive root processing element. Finally, these considerations result in recording the multiplication of '*rcount*' with '*group_size*' and with the size in bytes of a data element of type '*rtype*' for the number of bytes concerning the communication paths in the 1-to-N part of the N-to-1-to-N logical pattern.

☞ MPI_Alltoallv

A more universal version of the MPI_Alltoall function is the MPI_Alltoallv function:

int MPI_Alltoallv(**void** **sendbuf*, **int** **scounts*, **int** **sdisps*, MPI_Datatype *stype*,
void **recvb*, **int** **rcounts*, **int** **rdisps*,
MPI_Datatype *rtype*, MPI_Comm *comm*)

The effect of the MPI_Alltoallv function is similar to the execution of the MPI_Scatterv function with all participating processing element successively used as root processing element.

A participating processing element stores data elements of type '*stype*' in the array '*sendbuf*'. The value of '*scounts*[ii]' gives the number of data elements in '*sendbuf*' that is sent to the participating processing element with rank ii. The start position from which the data elements are sent to the processing element with rank ii is stored at '*sdisps*[ii]'. In the entrance hook, the number of bytes recorded with a send event equals the multiplication of the number of data elements sent to a participating processing element with the number of bytes needed to store a data element of type '*stype*'.

The received data elements are stored in the array '*recvbuf*'. The type of the received data elements is '*rtype*' and should be equal to '*stype*'. The data elements, which are received from the participating processing element with rank ii, are stored in '*recvb*' starting from position '*rdisps*[ii]'. The number of received data elements from the participating processing element with rank ii equals the value of '*rcounts*[ii]'. Therefore, the number of bytes recorded with a receive event from the participating processing element with rank ii is equal to the number of bytes needed to store a data element of type '*rtype*' multiplied by '*rcounts*[ii]'.

Because the number of data elements that is sent by the participating processing elements might differ, the only realistic logical pattern to record is the N-to-N logical pattern.

☞ MPI_Scan

The MPI_Scan function uses the following arguments:

int MPI_Scan(**void** **sendb*, **void** **recvb*, **int** *count*, MPI_Datatype *dt*,
MPI_Op *op*, MPI_Comm *comm*)

The arguments reveal that the MPI_Scan function is a special reduce function, using the function '*op*'.

Figure 5.1 explains how the associative function '*op*' is applied to the data elements stored in '*sendb*'. The results are stored in the array '*recvb*'. Both the results and input data elements are of type '*dt*'. The number of data elements used as input is given by '*count*', which is equal to the number of results stored in '*recvb*'. For all recorded events the number of bytes equals the multiplication of '*count*' with the size in bytes of a data element of type '*dt*'.

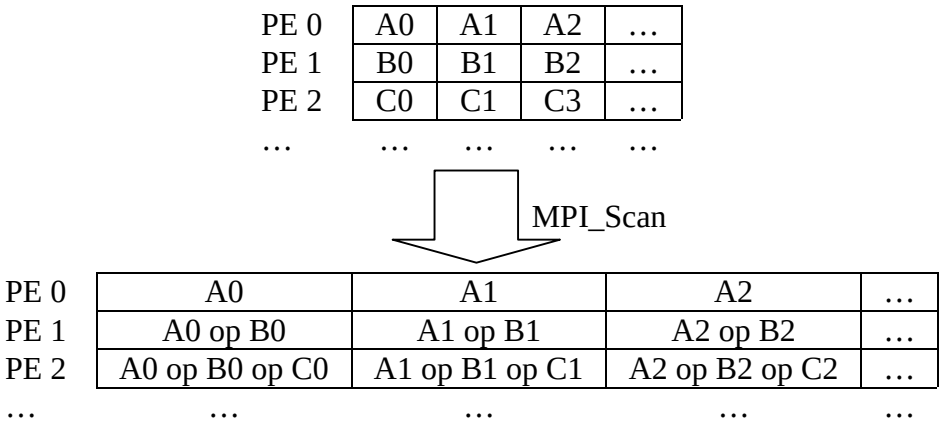

Figure 5.1. The result of the MPI_Scan function.

In figure 5.1, the participating processing element with rank i is denoted by PE i. The row indicated for a participating processing element represents the array '*sendb*' in the upper part of figure 5.1. In the lower part of the figure, a row indicated for a participating processing element represents the results stored in the '*recvb*' array after executing the MPI_Scan function. Note that the length of the arrays '*sendb*' and '*recvb*' might be different to the number of participating processing elements. It should be clear that the only realistic logical pattern to record is the N-to-N logical pattern.

N-to-M Collective Communication Functions with N-to-1-to-N default Pattern

The wrappers for three functions of the MPI library record an N-to-1-to-N logical pattern by default: MPI_Allgather, MPI_Allreduce and MPI_Reduce_scatter. The MPI_Allgather function implements the effect gained by executing a MPI_Gather function in which every participating processing element successively is the root processing element. MPI_Allreduce and MPI_Reduce_scatter are collective communication functions of which the effect is based on the effect of a reduce function.

☞ MPI_Allgather

The following arguments have to be used to call the MPI_Allgather function:

`int` MPI_Allgather(**`void`** **sendbuf*, **`int`** *scount*, MPI_Datatype *stype*, **`void`** **recvb*,
`int` *rcount*, MPI_Datatype *rtype*, MPI_Comm *comm*)

Every participating processing element stores data elements of type '*stype*' in the array '*sendbuf*'. The number of data elements that a participating processing element sends to the fictive root processing elements is given by '*scount*'. Therefore, the number of bytes recorded with a send event in the N-to-1 part of the N-to-1-to-N logical pattern is equal to the multiplication of '*scount*' with the number of bytes needed to store a data element of type '*stype*'.

The 1-to-N part of the N-to-1-to-N logical pattern represents the broadcast of '*group_size*' arrays '*recvb*', which are received by the fictive root processing element from all other participating processing elements. The variable '*group_size*' indicates the number of participating processing elements in the call of the MPI_Allgather function, including the fictive root processing element. The number of received data elements, which are of type '*rtype*' (should be equal to '*stype*'), in an array '*recvb*' is indicated by '*rcount*', which should be equal to '*scount*'. Therefore, the number of bytes recorded for the events in this part of the N-to-1-to-N logical pattern equals the multiplication of '*group_size*' with '*rcount*' and with the number of bytes needed to store a data element of type '*rtype*'. Note that, although not recorded by the N-to-1-to-N logical pattern, finally the array '*recvb*' is also changed for the fictive root processing element.

As long as the compile-time flag ALL_N_TO_N is not defined, the default N-to-1-to-N logical pattern is recorded if the function MPI_Allgather is in the '`list`' file. This logical pattern is recorded by default because the resulting array '*recvb*' is equal for all participating processing elements. However, if the compile-time flag ALL_N_TO_N is defined, an N-to-N logical pattern is recorded.

Every send event for the N-to-N logical pattern is recorded with '*scount*' times the size in bytes of a data element of type '*stype*' for the number of bytes concerning the communication path. The number of bytes recorded with the receive events for the N-to-N logical pattern is equal to the multiplication of '*rcount*' with the number of bytes needed to store a data element of type '*rtype*'.

☞ MPI_Allreduce

The arguments, needed for the MPI_Allreduce function, are:

`int` MPI_Allreduce(**`void`** **sendb*, **`void`** **recvb*, **`int`** *count*, MPI_Datatype *dt*,
MPI_Op *op*, MPI_Comm *comm*)

The effect of the MPI_Allreduce function is similar to executing a MPI_Reduce function with all participating processing elements successively used as root processing element.

In the N-to-1 part of the N-to-1-to-N logical pattern, all participating processing elements send '*count*' data elements of type '*dt*', stored in '*sendb*' to the fictive root processing element. After applying the associative function '*op*' on all the i^{th} data elements that are sent by every participating processing element, the fictive root processing element saves this result at '*recvb*[i]'. The results (of type '*dt*' again) stored in the array '*recvb*' of the fictive root processing element are broadcasted to all other participating processing elements in the 1-to-N part of the N-to-1-to-N logical pattern. Therefore, all events recorded for the N-to-1-to-N logical pattern have the multiplication of '*count*' with the size in bytes of a data element of type '*dt*' recorded for the number of bytes.

If the compile-time flag ALL_N_TO_N is defined, the wrapper for the MPI_Allreduce function records the alternative N-to-N logical pattern. The number of bytes recorded for all events in the N-to-N logical pattern is equal to the multiplication of '*count*' with the number of bytes needed to store a data element of type '*dt*'. Because an internal implementation of an N-to-1-to-N communication pattern for the MPI_Allreduce might be less complex, the N-to-1-to-N logical pattern is recorded by default.

☞ MPI_Reduce_scatter

The MPI_Reduce_scatter function uses the following arguments:

int MPI_Reduce_scatter(**void** **sendb*, **void** **recvb*, **int** **rcounts*, MPI_Datatype *dt*, MPI_Op *op*, MPI_Comm *comm*)

The functionality of the MPI_Reduce_scatter function consists of executing a MPI_Reduce function to a fictive root-processing element, after which a MPI_Scatterv function is executed. It should be clear that the N-to-1-to-N logical pattern is more realistic than the N-to-N logical pattern.

The participating processing elements all store data elements of type '*dt*' in the send buffer '*sendb*'. The number of data elements in '*sendb*' is equal to the sum of the elements '*rcounts*[ii]' for all ranks ii of the participating processing elements. This number is calculated in the wrapper for the MPI_Reduce_scatter function by a **for**-loop. The result is saved in the variable count. The recorded number of bytes for the send events in the N-to-1 part of the N-to-1-to-N logical pattern is equal to count multiplied by the size in bytes of a data element of type '*dt*'.

After calculating the result of executing the associative function '*op*' on all data elements that the participating processing operation store at '*sendb*[i]', the fictive root processing element saves the result at position i in a temporary (invisible) array. In the 1-to-N part of the logical pattern, the fictive root processing element scatters the data elements in this temporary array to all processing elements. However, only the number of data elements stored at '*rcounts*[ii]' is sent to the participating processing element with rank ii. The length of '*recvb*' of the participating processing element with rank ii must be equal to '*rcounts*[ii]'. The number of bytes recorded for the events in the 1-to-N part of the logical pattern, concerning the participating element with rank ii, equals the multiplication of the size in bytes of a data element of type '*dt*' with the number of data elements concerning the communication path from the fictive root processing element to the participating processing element with rank ii.

Although recording the N-to-1-to-N logical pattern for the MPI_Reduce_scatter function seams to be the most realistic communication pattern, it is possible to record an N-to-N logical pattern by defining the compile-time flag ALL_N_TO_N. There is, however, a problem if an N-to-N logical pattern is recorded. It is not clear what number of bytes should be recorded for the receive events in the exit hook for the N-to-N logical pattern because it is not known where the results of the function '*op*' are calculated. The current exit hook for the N-to-N logical pattern records '*rcounts*[ii]' multiplied by the number of bytes needed to store a data element of type '*dt*' for the number of bytes in a receive event.

5.2 Parallel Virtual Machine

To implement the wrappers for the PVM collective communication functions, some adaptations have to be made to the general frameworks, discussed in chapter 4. Because the PVM library does not use an initialisation function, it has to be checked whether the global variable PAT_collective is set correctly according to the value of the environment variable PAT_TRACE_COLLECTIVE.

Initially, the value of the global variable PAT_collective for the PVM library is set to –1. Because it is not known which communication function of the PVM library is used for the first communication in a parallel program, in every entrance hook for the PVM collective communication functions it has to be checked whether PAT_collective is already set correctly. Before checking PAT_collective for being 1 (in order to investigate whether the collective communication function must be recorded or not) an extra **`if`**-statement is added. This **`if`**-statement checks whether PAT_collective equals –1. If so, the function PAT_Set_collective() is called to set PAT_collective correctly according to PAT_TRACE_COLLECTIVE.

All communication functions of the PVM library use a tag value to coordinate the communication. Therefore, an argument named '*tag*' of the integer type is available for all PVM communication functions. This tag is recorded for every event in all wrappers for any communication function of the PVM library. Note that the PVM library does not have any collective communication function available for which an N-to-N or N-to-1-to-N logical pattern has to be recorded.

The PVM notation uses so-called groups to indicate a set of processing elements. To determine the number of processing elements in a group, the function pvm_gsize(**`char`** **group*) is used. The input argument '*group*' denotes the name of the group. On the Cray T3E a global group is predefined, containing all processing elements drawn on by the `mpprun` command. The name of the global group is a null name or the null **`char`**-pointer.

In parallel programs, processing element numbers are used to indicate communication paths. The standard PVM notation, however, uses the concept of task identifiers. The PVM implementation on the Cray T3E enables the use of processing element numbers instead of task identifiers. The function pvm_mytid() returns the task identity of the calling processing element. The function pvm_get_PE(**`int`** *tid*), available for the PVM implementation on the Cray T3E, returns the processing element number of the processing element with task identity '*tid*'. The task identity of a processing element is therefore similar to the rank of the processing element in a set of processing elements.

To determine the number of bytes needed to store a variable of a specific type, the PVM library defines the array pvm_type_size. The number of bytes needed to store a data element of type '*type*' is stored at pvm_type_size[*type*], where '*type*' is an integer value representing specific types defined by the PVM library.

1-to-N (Collective) Communication Functions

The PVM library enables the use of three functions that result in a 1-to-N pattern: pvm_bcast, pvm_mcast and pvm_scatter. Note that the pvm_bcast and pvm_mcast functions are no true collective communication functions. These two functions are used like a normal send function. This means that only special entrance hooks have to be written for these functions. A function starting with 'pvm_upk' records the receive events for the 1-to-N logical pattern.

☞ pvm_bcast

The pvm_bcast function uses the following arguments:

`int` pvm_bcast(**`char`** **group*, **`int`** *tag*)

The sending root processing element that calls the pvm_bcast function, broadcasts the message stored in the active communication buffer to all processing elements in '*group*'. Every receiving processing element in '*group*' uses the function pvm_recv or pvm_nrecv after which a function starting with 'pvm_upk' is called. When the receiving processing elements call a function starting with 'pvm_upk', a receive event is recorded with the correct number of received bytes. The wrappers for these functions are not discussed in this report.

The wrapper for the pvm_bcast function consists of a special entrance hook and a default exit hook. If checking whether the call of the function pvm_bcast has to be traced results in executing a **`for`**-loop to record N – 1 send events, the root processing first saves the location at which the first time stamp is recorded in timeptr. The function pvm_gsize is used to determine the number of processing elements in '*group*'. The variable ii, ranging from 0 to pvm_gsize(*group*) – 1, is used to record a send event to all processing elements in '*group*'.

Because no send event has to be recorded to the root processing element, the **`if`**-statement is used to exclude the root processing element from recording a send event to itself. To do this, the processing element number of the processing element with rank ii in '*group*' is compared with the processing element number of the processing element that called the entrance wrapper, being the root processing element.

Note that the number of bytes needed to store the message in the active communication buffer is not known until receiving the message. Therefore, the number of bytes recorded with a send event is set to 0. The wrappers of the functions starting with 'pvm_upk' determine the number of bytes received for the communication. After recording a send event with the EVENT_TRACE function, the time stamp is set equal to the value stored at timeptr.

☞ pvm_mcast

The arguments that need to be specified by the calling root processing element for the pvm_mcast function are:

`int` pvm_mcast(**`int`** **tids*, **`int`** *ntask*, **`int`** *tag*)

The array '*tids*' stores the task identities of '*ntask*' processing elements to which the message in the active communication buffer is sent by the calling root processing element. Every processing element of which the task identity is stored in the array '*tids*' receives the message using the function pvm_recv or pvm_nrecv after which a function starting with 'pvm_upk' is called. The wrappers for the functions starting with 'pvm_upk' record a receive event for all the processing elements of which the task identity is in '*tids*'. The wrappers for the functions starting with 'pvm_upk' are not discussed in this report.

The wrapper for the pvm_mcast function consists of a special entrance hook and a default exit hook. After checking whether the call of the pvm_mcast function must be traced or not, the root processing element executes a **`for`**-loop to record '*ntask*' send events. The **`if`**-statement is used to exclude the root processing element from recording a send event to itself.

Every send event is recorded with 0 for the number of bytes because the number of bytes needed to store the message in the active communication buffer is not known until receiving the message. The wrappers of the functions starting with 'pvm_upk', called after receiving the message, record the correct number of bytes for the communication.

☞ pvm_scatter

The pvm_scatter function is called with the following arguments:

int pvm_scatter(**void** *_recvbuf_, **void** *_sendbuf_, **int** _cnt_, **int** _type_, **int** _tag_, **char** *_group_, **int** _root_)

The collectively called pvm_scatter function evenly scatters the data elements stored in '*sendbuf*' from the root processing element successively to all participating processing elements in '*group*'. The number of data elements that is sent to every participating processing element is indicated by '*cnt*'. The data elements in the array '*sendbuf*' are of a type '*type*'. The value of '*root*' gives the rank of the root processing element in the set of participating processing elements '*group*'. Every participating processing element receives the data elements in the array '*recvbuf*', which is at least of length '*cnt*'. The length of '*sendbuf*' must at least be equal to the number of processing elements in '*group*' multiplied by '*cnt*'. Although not recorded by the 1-to-N logical pattern, the array '*recvbuf*' of the root processing element is also changed after calling the pvm_scatter function.

Because the internal implementation of the pvm_scatter function uses the pvm_psend and pvm_precv functions (See chapter 7), a global variable internal_event_trace_gate is introduced. The functions pvm_psend and pvm_precv are called after executing the entrance hook and before executing the exit hook. To disable tracing these functions, the global variable internal_event_trace_gate is set to 0 after checking whether the call of pvm_scatter must be traced or not. The wrappers for the functions pvm_psend and pvm_precv check the value of internal_event_trace_gate. If this global variable is 0, the functions pvm_psend and pvm_precv are not traced. Therefore, internal_event_trace_gate is initialised with the value 1.

After setting internal_event_trace_gate to 0, the root processing element is selected for executing a **for**-loop to record N – 1 send events. Note that the selecting **if**-statement also checks whether '*cnt*' is non-zero. In the **for**-loop, the root processing element is excluded from recording a send event to itself. The EVENT_TARCE function is used to record a send event with the number of bytes equal to the multiplication of '*cnt*' with the number of bytes needed to store a data element of type '*type*'.

Except for the root processing element, every participating processing element in '*group*' records a receive event in the exit hook. After recording the receive event, the value of internal_event_trace_gate is set back to 1 so that the functions pvm_psend and pvm_precv are traced if they are called after calling the pvm_scatter function if necessary.

N-to-1 Collective Communication Functions

In the PVM library, two functions for which the N-to-1 logical pattern is recorded are available: pvm_gather and pvm_reduce.

☞ pvm_gather

The arguments for the pvm_gather function are:

int pvm_gather(**void** *_recvbuf_, **void** *_sendbuf_, **int** _cnt_, **int** _type_, **int** _tag_, **char** *_group_, **int** _root_)

A participating processing element in '*group*' stores '*cnt*' data elements of '*type*' in the array '*sendbuf*'. The root processing element successively receives '*cnt*' data elements of '*type*' from every participating processing element in '*group*' into the array '*recvbuf*'. The rank of the root processing element in '*group*' is given by '*root*'. Note that the content of the array 'sendbuf' for the root processing element contributes to the final contents of '*recvbuf*'.

In the entrance hook, every participating processing element except the root processing element records a send event to the root processing element if '*cnt*' is non-zero. The number of bytes recorded for a send event equals the multiplication of '*cnt*' with the size in bytes of a data element of type '*type*'.

The root processing element is selected in the exit hook to record N – 1 receive events. The number of bytes recorded for a receive event equals the number of bytes recorded for a send event in the entrance hook.

☞ pvm_reduce

To call the pvm_reduce function, the following arguments have to be specified:

`int` pvm_reduce(**`void`** (**func*)(), **`void`** **buf*, **`int`** *cnt*, **`int`** *type*, **`int`** *tag*,
`char` **group*, **`int`** *root*)

For all participating processing elements in '*group*', the array '*buf*' stores '*cnt*' data elements of '*type*'. The root processing element of which the rank in the set of participating processing elements '*group*' is given by '*root*', executes the associative function '*func*' on the i^{th} element of all arrays '*buf*' and finally stores the result at '*buf*[i]'. So the array '*buf*' of the root processing element is overwritten after executing the pvm_reduce function.

In the entrance hook, the global variable internal_event_trace_gate is set to 0 to disable tracing of the functions pvm_psend and pvm_precv called by the internal implementation. All participating processing elements, except the root processing element, are selected to record a send event to the root processing element if '*cnt*' is non-zero. The number of bytes recorded for a send event equals the number of bytes needed to store a data element of '*type*' multiplied by '*cnt*'.

In the exit hook, the root processing element executes a **`for`**-loop to record a receive event from every other participating processing element. The number of bytes received from every participating processing elements is equal to the multiplication of '*cnt*' with the size in bytes of a data element of type '*type*'. After recording the receive events, the global variable internal_event_trace_gate is set back to 1.

5.3 Shared Memory

The implementations of the wrappers for the collective communication functions of the SHMEM library involve rather major adaptations of the general frameworks that are discussed in chapter 4. A reason for this is that the internal implementation of the collective communication functions consists of calls to SHMEM Point-to-Point communication functions. Chapter 7 discusses which Point-to-Point communication functions are used in the internal implementation of a specific collective communication function.

No initialising function has to be called before any of the functions in the SHMEM library can be used. However, before tracing a SHMEM collective communication function, it has to be checked whether the global variable PAT_collective is set correctly according to the value of the environment variable PAT_TRACE_COLLECTIVE. Because the initial value of PAT_collective is –1, in every entrance hook for the SHMEM collective communication functions an extra **`if`**-statement is executed to check whether PAT_collective equals –1. If so, the function PAT_Set_collective() is called to set PAT_collective correctly according to the value of PAT_TRACE_COLLECTIVE. Thereafter, the collective communication function is only traced if PAT_collective is equal to 1.

All SHMEM collective communication functions use three integer arguments to denote the set of participating processing elements in the call of the SHMEM collective communication function. The integer '*pestart*' is used to indicate the processing element with the smallest processing element number in the set of participating processing elements and '*pesize*' represents the total number of participating processing elements in the call of a collective communication function. The value of '*log_pestride*' is the logarithm to base 2 of the number of non-participating processing elements that is between two participating processing elements. The function PAT_pow(**`int`** *log_pestride*) is introduced to calculate the 2-based power of a given '*log_pestride*' number, resulting in the real number of non-participating processing elements that is between two participating processing elements in the call of a collective communication function.

The wrappers for the SHMEM collective communication functions are implemented using processing element numbers instead of rank numbers. The UNICOS/mk system function _my_pe() returns the processing element number of the calling processing element. To enable recording an N-to-1-to-N logical pattern, the function rank_to_PE(**`int`** *root,* **`int`** *pestart,* **`int`** *log_pestride*) is introduced for calculating the processing element number of a fictive root processing element. This function is also used in the wrappers for the 1-to-N collective communication functions to calculate the processing element number of the root processing element of which the rank in the set of participating processing elements is given by the integer argument '*peroot*'. Note that no N-to-1 collective communication functions are available in the SHMEM library.

The arguments of SHMEM collective communication functions include a synchronisation array '*psync*', containing data elements of type **`long`**. A wrapper does not have to investigate this array for correct functionality. The work array '*pwrk*', which has to be specified for several SHMEM collective communication functions, is also of no interest for the wrappers. The SHMEM library defines the types _CNST **`void`** and _SIZE_T to indicate a source array and its length respectively.

The wrappers of many SHMEM communication functions have the same structure. Collective communication functions, of which the wrappers have the same structure, differ only in their names and the type of the data elements on which they act. Although the number of SHMEM communication functions is rather large, their wrappers are generated automatically by using templates for the different wrapper structures. For every communication method of which functions are available in the SHMEM library, tables are given of the functions that use the same wrapper template. These tables also depict the aliases and the Fortran equivalents of the functions.

Because SHMEM Point-to-Point communication functions are called by the internal implementation of the SHMEM collective communication functions, the (global) variable EVENT_TRACE_GATE is used to disable tracing of any communication that occurs between the execution of the entrance and exit hook of a collective communication function. Therefore, EVENT_TRACE_GATE is not set back to 1 after executing the entrance hook. This, however, introduces a problem for executing the exit hook. The exit hook for a collective communication function must be executed if the value of EVENT_TRACE_GATE was 1 before executing the entrance hook of that function. The introduction of a global variable no_internal_event_trace_gate, which is initialised to 0, solves this problem. The exact use of no_internal_event_trace_gate is clarified at discussing the wrapper templates.

1-to-N Collective Communication Functions

The two 1-to-N collective communication functions of the SHMEM library, for which special wrappers are available, are given in table 5.1.

Wrapper for Function:	'*bytes*':	Alias:	Fortran Equivalents:
shmem_broadcast32	4		SHMEM_BROADCAST4 SHMEM_BROADCAST32
shmem_broadcast64	8	shmem_broadcast	SHMEM_BROADCAST SHMEM_BROADCAST8 SHMEM_BROADCAST64

Table 5.1. Collective communication functions of which the wrappers are generated with the shmem_*broadcast* template.

The wrappers for the broadcast functions shmem_broadcast32 and shmem_broadcast64 are generated using the shmem_*broadcast* template. The value of '*bytes*' indicates the number of bytes needed to store a data element on which the shmem_*broadcast* function acts.

☞ shmem_*broadcast*

The shmem_*broadcast* functions use the following arguments:

void shmem_*broadcast*(**void** **trg*, _CNST **void** **src*, _SIZE_T *len*, **int** *peroot*,
int *pestart*, **int** *log_pestride*, **int** *pesize*,
long **psync*)

The sending root processing element, of which the rank in the set of participating processing elements is available via the argument '*peroot*', stores '*len*' data elements in the send array '*src*'. The number of bytes needed to store a single data element in '*src*' is known via the name of the broadcast function and is indicated by '*bytes*', see table 5.1. Every participating processing element other than the root processing element receives the data elements in the receive array '*trg*'. The length of this array should at least be equal to '*len*'.

After checking whether PAT_collective is 1, EVENT_TRACE_GATE is set to 0 in the entrance hook to disable tracing of SHMEM Point-to-Point communication functions that are called by the internal implementations of the shmem_*broadcast* functions. The global variable no_internal_event_trace_gate is set to 1 to indicate that the root processing element records N – 1 send events in the entrance hook. So, the variable no_internal_event_trace_gate stores that PAT_collective is 1 and EVENT_TRACE_GATE was 1.

To select the root processing element for executing a **for**-loop to record N – 1 send events, the result of the call of _my_pe() is compared with the variable global_root, which stores the processing element number of the root processing element. Because the number of data elements that is sent to all participating processing elements other than the root processing element equals '*len*', the selecting **if**-statement also checks whether '*len*' is non-zero.

After selecting the root processing element for executing a **for**-loop to record N – 1 send events, the local variable dist is used to save the real number of non-participating processing elements that is between two participating processing elements. The root processing element number of every next participating processing element, to which a send event is recorded, is calculated by adding dist to the counter ii. The counter ii ranges from '*pestart*' to ('*pestart*' + dist * '*pesize*') with steps of size dist and therefore runs through processing element numbers. The root processing element is excluded from recording a send event to itself. The number of bytes recorded with a send event is equal to the multiplication of '*len*' with '*bytes*'.

Note that EVENT_TRACE_GATE is not set back to 1 in the entrance hook. In order to know whether a receive event must be recorded by every participating processing element other than the root processing element in the exit hook, an **if**-statement checks whether EVENT_TRACE_GATE is 0 and no_internal_event_trace_gate is 1. If so, the variable global_root is used to store the processing element number of the root processing element. An **if**-statement is used to select all participating processing elements other than the root processing element for recording a receive event if '*len*' is non-zero. The number of bytes that is recorded with a receive event equals the multiplication of '*len*' with '*bytes*'. If receive events are recorded in the exit hook, EVENT_TRACE_GATE has to be set back to 1. A final **if**-statement is used to call the EVENT_TRACE_SHORT_EXIT function only if EVENT_TRACE_GATE is set back to 1.

N-to-M Collective Communication Functions with N-to-N default Pattern

The wrappers for the collect functions shmem_collect32 and shmem_collect64 are generated using the shmem_*collect* template. Table 5.2 shows all collect functions of the SHMEM library. The value of '*bytes*' indicates the size in bytes of a data element on which the function acts.

Wrapper for Function:	'*bytes*':	Alias:	Fortran Equivalent(s):
shmem_collect32	4		SHMEM_COLLECT4
shmem_collect64	8	shmem_collect	SHMEM_COLLECT SHMEM_COLLECT8

Table 5.2. Collective communication functions of which the wrappers are generated with the shmem_*collect* template.

☞ shmem_*collect*

The shmem_*collect* functions, which are a more universal version of the shmem_*fcollect* functions, use the following arguments:

void shmem_*collect*(**void** **trg*, _CNST **void** **src*, _SIZE_T *len*, **int** *pestart*,
int *log_pestride*, **int** *pesize*, **long** **psync*)

Every participating processing element in the call of a shmem_*collect* function successively sends the data elements stored in the array '*src*' to all other participating processing elements with an increasing rank number in the set of all participating processing elements. The number of data elements in '*src*' is indicated by '*len*' and may be different for every participating processing element. A participating processing element successively receives the data elements in the array '*trg*'. The length of '*trg*' must at least be equal to the sum of the values of all variables '*len*' of all participating processing elements.

The internal implementations of the shmem_*collect* functions use the corresponding shmem_*fcollect* functions, see chapter 7. To correctly implement the wrappers for the shmem_*collect* functions, the shmem_fcollect function has to be called too. Because the global variable no_internal_event_trace_gate is already used to disable recording internally called Point-to-Point communication functions in the shmem_*fcollect* functions, an extra global variable is necessarily introduced: no_fcollect_event_trace_gate. The value of this variable is 0 after initialisation.

To record the default N-to-N logical pattern, the global variable no_fcollect_event_trace_gate is set to 1 after checking that PAT_collective is 1 in the entrance hook. This enables recording receive events during the execution of the exit hook for the N-to-N logical pattern after setting EVENT_TRACE_GATE to 0 in the entrance hook. For every participating processing element it is checked whether '*len*' is non-zero. If so, a send event is recorded to all other participating processing elements. The number of bytes recorded with a send event is equal to the multiplication of '*len*' with '*bytes*'. Note that the counter ii in the **`for`**-loop runs trough the processing element numbers of all participating processing elements.

In the exit hook of the N-to-N default logical pattern, it is checked whether EVENT_TRACE_GATE is 0 and no_fcollect_event_trace_gate is 1 before recording receive events. Every participating processing element in the call of a shmem_*collect* function only knows the value of his variable '*len*'. However, to record the correct number of bytes received from every other participating processing element, every participating processing element must know the values of all variables '*len*' of all participating processing elements.

The shmem_fcollect function is used to gather the values of all integer variables '*len*' in an array rcounts. The value of the variable '*len*' of the participating processing element with rank jj in the set of participating processing elements is saved at rcounts[jj].

A **`for`**-loop is executed by every participating processing element to record N – 1 receive events from all other participating processing elements. The counter ii, which runs through the processing element numbers, is initialised to '*pestart*', whereas the counter jj, running trough the rank numbers of the participating processing elements, is initialised to 0. The processing element with processing element number ii has rank jj in the set of all participating processing elements. Therefore, the number of bytes received from the participating processing element with processing element number ii is stored at rcounts[jj]. The number of bytes recorded for a receive event is equal to the multiplication of '*bytes*' with the number of data elements that is received from the concerning participating processing element. After freeing the address space of the rcounts array, EVENT_TRACE_GATE is set back to 1.

Because an internal implementation that results in an N-to-N communication pattern might be rather uncomplicated, the N-to-N logical pattern is recorded by default.

Because the content of the '*trg*' array for all participating processing elements is the same after executing a shmem_*collect* function, the alternative N-to-1-to-N logical pattern is recorded if the compile-time flag ALL_N_TO_1_TO_N is defined.

To record the N-to-1-to-N logical pattern, the processing element, of which the rank number in the set of participating processing elements equals '*pesize*' divided by 2, is used as fictive root processing element. The function rank_to_PE calculates the processing element number of the fictive root processing element. After setting the value of no_fcollect_event_trace_gate to 1 in the entrance hook, every participating processing element other than the fictive root processing element records a send event to the fictive root processing element. The recorded number of bytes is equal to the multiplication of '*len*' with '*bytes*'.

In the exit hook, the same fictive root processing element is selected as in the entrance hook. rcounts[jj] stores the value of the variable '*len*' of the processing element with rank jj in the set of participating processing elements after calling the shmem_fcollect function. Note that only the fictive root processing element needs all elements of rcounts for recording the correct number of bytes received from every other participating processing element in the N-to-1 part of the N-to-1-to-N logical pattern. For the 1-to-N part of the N-to-1-to-N logical pattern, only the total number of data elements in all the send arrays '*src*' of all participating processing elements, including the fictive root processing element, has to be known. The local variable total stores this total number of data elements.

Note that the variable total has to be checked for being non-zero both by the fictive root processing element and all other participating processing elements because the value of total is unavailable from the arguments of the shmem_*collect* functions. If total equals zero, no communication paths have to be recorded at all because total is only zero if the value of rcounts[jj] is zero for all ranks jj.

N-to-M Collective Communication Functions with N-to-1-to-N default Pattern

Table 5.3 and 5.4 give the functions of the SHMEM library for which the N-to-1-to-N logical pattern is recorded by default. Although the number of different functions is rather large, only two different templates are used to generate the wrappers for these functions. Because the two wrapper templates differ only in the way the number of bytes, which is needed to store the data elements on which the function acts, is given by the name of the concerning functions, they are discussed simultaneously.

Wrapper for Function:	'*bytes*':	Alias:	Fortran Equivalent(s):
shmem_fcollect32	4		SHMEM_FCOLLECT4
shmem_fcollect64	8	shmem_fcollect	SHMEM_FCOLLECT SHMEM_FCOLLECT8

Table 5.3. Collective communication functions of which the wrappers are generated with the shmem_*fcollect* template.

☞ shmem_*fcollect*

For the shmem_*fcollect* functions, the following arguments need to be specified:

void shmem_*fcollect*(**void** *trg, _CNST **void** *src, _SIZE_T *len*, **int** *pestart*,
int *log_pestride*, **int** *pesize*, **long** *psync*)

The shmem_*fcollect* functions are used to collect the data elements stored in the array '*src*' by every participating processing element. The number of data elements in '*src*' is equal for every participating processing element and is indicated by '*len*'. Every participating processing element successively receives the data elements in the array '*trg*'. The number of data elements saved in '*trg*' equals '*len*' multiplied by '*pesize*'.

Wrapper for Function:	Aliases:	Fortran Equivalent:
shmem_complexd_prod_to_all		SHMEM_COMP8_PROD_TO_ALL
shmem_complexd_sum_to_all		SHMEM_COMP8_SUM_TO_ALL
shmem_complexf_prod_to_all		SHMEM_COMP4_PROD_TO_ALL
shmem_complexf_sum_to_all		SHMEM_COMP4_SUM_TO_ALL
shmem_double_max_to_all		SHMEM_REAL8_MAX_TO_ALL
shmem_double_min_to_all		SHMEM_REAL8_MIN_TO_ALL
shmem_double_prod_to_all		SHMEM_REAL8_PROD_TO_ALL
shmem_double_sum_to_all		SHMEM_REAL8_SUM_TO_ALL
shmem_float_max_to_all		SHMEM_REAL4_MAX_TO_ALL
shmem_float_min_to_all		SHMEM_REAL4_MIN_TO_ALL
shmem_float_prod_to_all		SHMEM_REAL4_PROD_TO_ALL
shmem_float_sum_to_all		SHMEM_REAL4_SUM_TO_ALL
shmem_int_and_to_all	shmem_long_and_to_all shmem_longlong_and_to_all	SHMEM_INT8_AND_TO_ALL
shmem_int_max_to_all	shmem_long_max_to_all shmem_longlong_max_to_all	SHMEM_INT8_MAX_TO_ALL
shmem_int_min_to_all	shmem_long_min_to_all shmem_longlong_min_to_all	SHMEM_INT8_MIN_TO_ALL
shmem_int_or_to_all	shmem_long_or_to_all shmem_longlong_or_to_all	SHMEM_INT8_OR_TO_ALL
shmem_int_prod_to_all	shmem_long_prod_to_all shmem_longlong_prod_to_all	SHMEM_INT8_PROD_TO_ALL
shmem_int_sum_to_all	shmem_long_sum_to_all shmem_longlong_sum_to_all	SHMEM_INT8_SUM_TO_ALL
shmem_int_xor_to_all	shmem_long_xor_to_all shmem_longlong_xor_to_all	SHMEM_INT8_XOR_TO_ALL
shmem_short_and_to_all		SHMEM_INT4_AND_TO_ALL
shmem_short_max_to_all		SHMEM_INT4_MAX_TO_ALL
shmem_short_min_to_all		SHMEM_INT4_MIN_TO_ALL
shmem_short_or_to_all		SHMEM_INT4_OR_TO_ALL
shmem_short_prod_to_all		SHMEM_INT4_PROD_TO_ALL
shmem_short_sum_to_all		SHMEM_INT4_SUM_TO_ALL
shmem_short_xor_to_all		SHMEM_INT4_XOR_TO_ALL

Table 5.4. Collective communication functions of which the wrappers are generated with the shmem_*type_func*_to_all template.

☞ shmem_*type_func*_to_all

The arguments for the shmem_*type_func*_to_all reduce functions are:

void shmem_*type_func*_to_all(**void** **trg*, **void** **src*, _SIZE_T *len*, **int** *pestart*, **int** *log_pestride*, **int** *pesize*, *type* **pwrk*, **long** **psync*)

Every participating processing element in the call of a shmem_*type_func*_to_all function stores '*len*' data elements of type '*type*' in the array '*src*'. The i^{th} data element of the '*src*' arrays of all participating processing elements contributes to the result calculated with the associative function '*func*' and stored at the i^{th} position of the receiving array '*trg*'. After executing the shmem_*type_func*_to_all function, the arrays '*trg*' in all participating processing elements are equal. The length of the array '*trg*' should at least be equal to '*len*'.

In table 5.3, '*bytes*' gives the size in bytes of the data elements on which the shmem_*fcollect* function acts. The number of bytes needed to store a data element on which the shmem_*type_func*_to_all act, is given by **`sizeof`**(*type*).

After checking whether PAT_collective equals 1 in the entrance hook of the templates used to generate the wrappers for the shmem_*fcollect* and shmem_*type_func*_to_all functions, EVENT_TRACE_GATE is set to 0. The global variable no_internal_event_trace_gate is set to 1 to disable tracing of any internally called Point-to-Point communication functions of the SHMEM library. The rank number of the fictive root processing element is equal to '*pesize*' divided by 2. Any participating processing element other than the fictive root processing element for which '*len*' is non-zero is selected to record a send event to the fictive root processing element.

In the exit hook, the EVENT_TRACE_GATE must be 0 and no_internal_event_trace_gate must be 1 to record the receive events for the N-to-1 part and to record the 1-to-N part of the N-to-1-to-N logical pattern. The fictive root processing element is selected to record a receive event from all other participating processing elements first. The number of bytes recorded for a receive event is equal to the number of bytes needed to store a data element on which the function acts multiplied by '*len*'. A second **`for`**-loop is executed by the fictive root processing element to record N - 1 send events for the 1-to-N part of the N-to-1-to-N logical pattern. The number of bytes recorded for every send event is equal to the size in bytes of a data element on which the function acts multiplied by '*len*' and '*pesize*'. All processing elements other than the fictive root processing element record a receive event for the 1-to-N part of the N-to-1-to-N logical pattern in the exit hook. Thereafter is EVENT_TRACE_GATE set back to 1.

If the compile-time flag ALL_N_TO_N is defined, the alternative N-to-N logical pattern is recorded for the shmem_*fcollect* and shmem_*type_func*_to_all functions. In the entrance hook, every participating processing element for which '*len*' is non-zero is selected to record N – 1 send events to all other participating processing elements. The number of bytes recorded with a send event is equal to the multiplication of '*len*' with the size in bytes of a data element on which the function acts. The exit hook for the N-to-N logical pattern consists of recording N – 1 receive events by all participating processing elements. The number of bytes recorded for the receive events equals the multiplication of '*len*' with the size in bytes of a data element on which the function acts.

Atomic Communication Functions

Although the atomic communication functions, available in the SHMEM library, are no (true) collective communication functions, more than one communication path may occur between two processing elements. Only two kinds of atomic communication functions are available: conditional and unconditional atomic communication functions. The patterns that are recorded by the wrappers of these two kinds of atomic functions are depicted in figure 5.2.

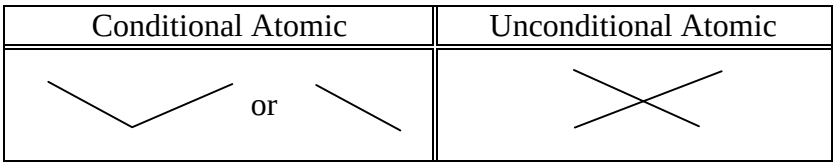

Figure 5.2. The patterns that are recorded for SHMEM atomic communication functions.

If a processing element calls an atomic communication function, a variable that is available on the remote processing element is inspected and the contents eventually changed. Although an atomic communication function involves two processing elements, the remote processing element is not informed if the atomic communication function changes the remote processing element's memory contents. Conditional atomic communication functions first check whether a specific condition holds before changing the remote processing element's memory contents.

Because not both participating processing elements, which contribute to the effect of an atomic communication function call, execute the atomic communication function and therefore its wrapper, special send and receive attributes are necessary to record send and receive events. The type of a recorded event for the atomic communication functions is PAT_TRACE_RPUT, PAT_TRACE_PUT, PAT_TRACE_RGET or PAT_TRACE_GET. To record a complete communication path from a remote processing element to the processing element that called the atomic communication function, a send event of type PAT_TRACE_RPUT must be combined with a receive event of type PAT_TRACE_GET. Recording a communication path from the calling processing element to the remote processing element consists of recording a send event of type PAT_TRACE_PUT combined with a receive event of type PAT_TRACE_RGET.

Conditional Atomic Communication Functions

Table 5.5 gives the conditional atomic communication functions of the SHMEM library for which the shmem_*type_cfunc* template is used to generate their wrappers. The number of bytes that must be recorded for the communication events occurring in the call of the C function shmem_int_cswap and shmem_short_cswap is respectively **`sizeof`**(**`int`**) and **`sizeof`**(**`short`**). For the SHMEM_INT4_CSWAP and the SHMEM_INT8_CSWAP Fortran functions, the number of bytes recorded equals 4 respectively 8.

Wrapper for Function:	Alias:
shmem_int_cswap	shmem_long_cswap
shmem_short_cswap	
SHMEM_INT4_CSWAP	
SHMEM_INT8_CSWAP	

Table 5.5. Conditional atomic communication functions of which the wrappers are generated with the shmem_*type_cfunc* template.

☞ shmem_*type_cfunc*

The conditional atomic communication functions for which the wrappers are generated with the shmem_*type_cfunc* template have the following arguments:

type shmem_*type_cfunc*(*type* **addr*, *type cond*, *type value*, **`int`** *pe*)

The pointer '*addr*' that is available for the remote processing element of which the processing element number is indicated by '*pe*', points to a data element of the type '*type*' on which the called shmem_*type_cfunc* function acts. If the value of '*cond*' is equal to the value of that data element, the value of the argument '*value*' is stored at '*addr*' in the remote processing element's memory. Next to this, the contents of the data element to which '*addr*' points is returned by the shmem_*type_cfunc* function. If '*cond*' is not equal to the data element pointed to by '*addr*', no memory contents are changed.

In the entrance hook for the shmem_*type_cfunc* functions, it is first checked whether EVENT_TRACE_GATE is 1. If so, the time at which the entrance hook is executed is saved in the global variable starttime after calling the EVENT_TRACE_SHORT_ENTRY function. Thereafter, a send event from the remote processing element is recorded by recording a send event of type PAT_TRACE_RPUT.

To enable the investigation of whether the condition for changing memory contents is true, the function pat_get_int_ret() is called first in the exit hook. Calling this function results in saving the return value of the shmem_*type_cfunc* function into the local variable result. Because recording the first part of the communication paths for the shmem_*type_cfunc* must always be recorded, a receive event of type PAT_TRACE_GET is recorded for the calling processing element.

If the return value of the shmem_*type_cfunc* is equal to '*cond*', the second part of the communication paths must also be recorded. However, the time of the recorded receive event for the calling processing element must first be set to the arithmetic mean of starttime and the time at which the exit hook is executed. Thereafter, a send event of type PAT_TRACE_PUT from the calling processing element to the remote processing element is recorded. The time for this send event is also manipulated to get the correct communication pattern. Finally, a receive event is recorded for the remote processing element by using the event type PAT_TRACE_RGET.

Unconditional Atomic Communication Functions

The template shmem_*type_func* is used to generate the wrappers of the unconditional atomic communication functions of table 5.6, see next page. For the exact effect of a shmem_*type_func* function, the reader is referred to [`man shmem_type_func`]. For the C functions of table 5.6, the number of bytes recorded for the communication events equals **`sizeof`**(*type*), whereas for the Fortran functions the number of bytes recorded equals the number (4 or 8) in the function's name.

☞ shmem_*type_func*

For the shmem_*type_func* functions, the following arguments must be specified:

type shmem_*type_func*(*type* *****addr*, type mask, type value*, **`int`** *pe*)

Note that the argument '*mask*' is only available for the functions '*func*' that include the letter 'm' or 'M'. Two simultaneous communication paths are recorded for the shmem_*type_func* functions: a communication path from the calling processing element to the remote processing element and vice versa.

Wrapper for Function:	Aliases:
shmem_double_swap	
shmem_float_swap	
shmem_int_mswap	shmem_long_mswap
shmem_int_swap	shmem_swap shmem_long_swap
shmem_short_add	
shmem_short_fadd	
shmem_short_finc	
shmem_short_inc	
shmem_short_mswap	
shmem_short_swap	
SHMEM_INT4_ADD	
SHMEM_INT4_FADD	
SHMEM_INT4_FINC	
SHMEM_INT4_INC	
SHMEM_INT4_MSWAP	
SHMEM_INT4_SWAP	
SHMEM_INT8_MSWAP	
SHMEM_INT8_SWAP	SHMEM_SWAP
SHMEM_REAL4_SWAP	
SHMEM_REAL8_SWAP	

Table 5.6. Unconditional atomic communication functions of which the wrappers are generated with the shmem_*type_func* template.

The entrance hook for the shmem_*type_func* functions consists of recording a send event of type PAT_TRACE_RPUT from the remote processing element to the calling processing element and a send event of type PAT_TRACE_PUT from the calling processing element to the calling processing element. In the exit hook, corresponding receive events are recorded to complete the communication paths for the shmem_*type_func* functions.

6 Testing the Wrappers

In this chapter it is described how the wrappers for the collective and atomic communication functions of the message passing libraries are tested. The main subject is the determination of the arguments for a communication function and the way in which the effects are checked for being correct. The source code of every test program is given in the appendix.

After instrumentation of a test program, the program ‘piftest’ is used to investigate the recorded event trace. The program VAMPIR by Pallas is used for a graphical visualisation of the communication paths recorded during the execution of a test program.

In all test programs, it is checked whether the total number of all processing elements used to execute the test program is more than one. If not, the test program stops the execution by giving a warning message. In many test programs, the total number of participating processing elements is used to parameterise the number of data elements that is sent and received for a communication function. If a test program executes a communication function incorrectly, error messages are generated to give an indication of what error has occurred.

6.1 Message Passing Interface

The programs discussed in this section test the collective communication functions of the MPI library without defining extra communicators. However, the wrappers for the collective communication functions support the use of different communicators. Note that the set of all processing elements drawn on by the `mpprun` command is also a communicator. This communicator is named MPI_COMM_WORLD and is defined in the MPI library. The effect of not defining extra communicators is that the rank of a processing element in the total set of all participating processing elements in MPI_COMM_WORLD equals its processing element number. A test program is available that tests a number of MPI collective communication functions using extra communicators, see the additional test programs in the appendix.

All test programs that are available for the collective communication functions of the MPI library result in recording the expected event trace if analysed with PAT. This includes the recorded number of bytes for an event and the resulting graphical visualisation of the logical pattern. Therefore, it can be concluded that the wrappers for the collective communication functions of the MPI library are implemented correctly.

1-to-N Collective Communication Functions

☞ MPI_Bcast

In the test program for the MPI_Bcast function, the variable id is used to store the rank of a processing element in the set of all participating processing elements. The variable numprocs gives the number of all processing elements in the communicator MPI_COMM_WORLD, drawn on by the `mpprun` command. The root processing element for the MPI_Bcast function, represented by master, is the processing element with the largest rank. The root processing element broadcasts the single data element buf of the type integer.

The value of buf is set to 255 for the root processing element only. After executing the MPI_Bcast function by all processing elements, it has to be checked whether the value of buf is equal to 255 for all processing elements. If so, no broadcast errors have occurred.

☞ MPI_Scatter

To test the MPI_Scatter function, a send array (named sbuf) of numprocs integers is declared for the root processing element master in the test program for the MPI_Scatter function. The variable numprocs gives the number of participating processing elements in MPI_COMM_WORLD. The integer stored at sbuf[i] is set to the result of i + 25.

Every participating processing element receives one integer from the root processing element in the receive buffer rbuf. The value of rbuf that must be valid for a processing element is equal to its rank in MPI_COMM_WORLD added with 25. The rank of a processing element is available via the variable id. Because rbuf is set to 17 for the root processing element before the MPI_Scatter function is called, it is checked that the value of rbuf changes for the root processing element after calling MPI_Scatter.

☞ MPI_Scatterv

In the test program for the MPI_Scatterv function, the root processing element, of which the rank in MPI_COMM_WORLD is saved in master, scatters i + 1 integers, ranging from 0 to i, to the participating processing element with rank i in MPI_COMM_WORLD. Therefore, the root processing element declares and initialises the array sbuf, containing the correct integer values that have to be sent to the different participating processing elements. The array scounts stores at position i the number of integers that has to be sent to the participating processing element with rank i. At displ[i] the position in sbuf is saved from which the root processing element starts sending integers to the processing element with rank i.

After calling the MPI_Scatterv function, all participating processing elements in MPI_COMM_WORLD check whether the contents of the receive buffer rbuf ranges from 0 to the rank of the concerning processing element. If so, no scatter errors have occurred. Note that the number of elements in rbuf for a processing element number is equal to its rank in MPI_COMM_WORLD plus 1.

N-to-1 Collective Communication Functions

☞ MPI_Gather

The integer id stores the rank of a processing element in the set of participating processing elements MPI_COMM_WORLD in the test program for the MPI_Gather function. The total number of processing elements in MPI_COMM_WORLD is saved in numprocs. The root processing element, which is indicated by the variable master, is the processing element with the largest rank in the set of participating processing elements.

The root processing element successively gathers the rank value of every participating processing elements and saves them in the receive array rbuf. After calling the MPI_Gather function, the value stored at rbuf[i] has to be equal to i, otherwise the MPI_Gather function was not correctly used.

☞ MPI_Gatherv

The participating processing element with rank i in MPI_COMM_WORLD, saved in the integer id, stores i + 1 integers ranging from 0 to i in the send buffer sbuf. After calling the MPI_Gatherv function, the root processing element, of which the rank is indicated by master, successively receives the integers in sbuf from all participating processing elements into the receive buffer rbuf. This buffer is initialised to be able to hold the correct number of integers. At position rcounts[i] the root processing element stores the number of integers that has to be received from the participating processing element with rank i in MPI_COMM_WORLD. The positions from which the received integers are stored in rbuf are initialised in displ.

After calling the MPI_Gatherv function, the root processing element checks whether i + 1 integers, ranging from 0 to i, are received in rbuf, starting from position displ[i], from the processing element with rank i in MPI_COMM_WORLD. If so, no gather errors have occurred.

☞ MPI_Reduce

Every participating processing element in MPI_COMM_WORLD stores numprocs times its rank number in the send buffer sbuf. The value of numprocs is equal to the number of processing elements in MPI_COMM_WORLD. After calling the MPI_Reduce function, the root processing element, of which the rank is saved in master, received numprocs results from executing the MPI_SUM addition function on the integers stored in all buffers sbuf. These results are stored in rbuf. The value of rbuf[i] equals the result of executing the MPI_SUM function on the integers saved by every participating processing element at sbuf[i].

The result stored at rbuf[i] has to be equal to the sum of the ranks ranging from 0 to numprocs – 1, for all i. If this is found to be true, no reduce errors are generated.

N-to-M Collective Communication Functions with N-to-N (default) Pattern

☞ MPI_Allgatherv

In the test program for the MPI_Allgatherv function, the processing element with rank i in the set of participating processing elements, indicated by MPI_COMM_WORLD, stores i + 1 integers ranging from 0 to i in the send buffer sbuf. Calling the MPI_Allgatherv function results in receiving integers in the array rbuf. The number of integers that is received from the processing element with rank i, saved at rcounts[i], equals i + 1. The position from which every participating processing element stores the received integers of the processing element with rank i is given by displ[i].

No gather errors occur if all participating processing elements received i + 1 integers from the processing element with rank i, ranging from 0 to i and stored from position displ[i] in the array rbuf. The value of i ranges from 0 to the total number of participating processing elements.

☞ MPI_Alltoall

The variable numprocs represents the number of participating processing elements in the test program for the MPI_Alltoall collective communication function.

Every participating processing element declares a receive and send array, rbuf and sbuf respectively, to store numprocs integers. The send buffer sbuf for the MPI_Alltoall function is initialised with integers ranging from 0 to numprocs – 1.

The effect of calling the MPI_Alltoall function is that the participating processing element with rank i has numprocs times its rank stored in the array rbuf. If this is not true for any participating processing element, an error is printed.

☞ MPI_Alltoallv

All participating processing elements in MPI_COMM_WORLD store integers in the send buffer sbuf. The number of integers stored in sbuf is equal to the sum of all results of i + 1, where i ranges from 0 to numprocs - 1. The variable numprocs indicates the number of participating processing element in MPI_COMM_WORLD. The array scounts stores at position i the number of integers in sbuf that is send to the processing element with rank i by the MPI_Alltoallv function. The value of sdispl[i] indicates from which position in sbuf the integers that have to be sent to the processing element with rank i are stored.

The number of integers that is received by the processing element with rank i from all other processing elements is equal to its rank + 1, stored in rcounts[i]. Therefore, the length of the receive buffer rbuf of the participating processing element with rank i equals the multiplication of (i + 1) with numprocs. The position in rbuf from which integers, received from the participating processing element with rank i, are saved is indicated by rdispl[i].

After calling the MPI_Alltoallv function, the participating processing element with rank i received numprocs times integers ranging from 0 to i. If so, no error messages are generated.

☞ MPI_Scan

In the test program for the MPI_Scan function, every participating processing element in MPI_COMM_WORLD saves its rank number numprocs times in the array sbuf. The variable numprocs indicates the total number of participating processing elements. According to figure 5.1, a receive buffer rbuf of length numprocs is needed to store the results after calling the MPI_Scan function. The MPI_SUM addition function is used in the test program for the MPI_Scan function.

The result of calling the MPI_Scan function is that every element of the receive buffer rbuf of the processing element with rank i must equal the sum of all i^{th} components of the sbuf arrays, sent by all other participating processing elements. If this is not valid for a processing element, a scan error is printed.

N-to-M Collective Communication Functions with N-to-1-to-N default Pattern

☞ MPI_Allgather

Every participating processing element sends its rank number, stored in the variable id, to all other participating processing elements. To receive these rank number integers from all other participating processing elements, a processing element declares the array rbuf to store the numprocs received integers. The total number of participating processing elements in MPI_COMM_WORLD is stored in numprocs.

After calling the MPI_Allgather function, the integer element received in rbuf[i] equals i, for all i ranging from 0 to numprocs – 1. If there is any participating processing element for which this is not valid, it generates a gather error.

☞ MPI_Allreduce

To test the MPI_Allreduce function, every participating processing element stores its rank number in the numprocs entries of the array sbuf. The variable numprocs represents the number of participating processing elements in MPI_COMM_WORLD. The rank of a processing element is available via the variable id. To enable storing the numprocs results gained after calling the MPI_Allreduce function, every participating processing element declares an array rbuf.

After calling the MPI_Allreduce function, the array rbuf stores at position i the sum of all received integers stored by all participating processing elements at sbuf[i]. Note that all elements of rbuf are the same after the call of MPI_Allreduce, for all participating processing elements. If this is not true, a reduce error is generated.

☞ MPI_Reduce_scatter

To test the MPI_Reduce_scatter function, every participating processing element in MPI_COMM_WORLD introduces a send buffer sbuf of length numprocs. The variable numprocs represents the number of all participating processing elements in MPI_COMM_WORLD. All entries of the array sbuf are initialised with the rank of the concerning processing element, which is available via the variable id. A participating processing element receives the results of the summing MPI_Reduce_scatter call, consisting of a single integer, in the variable rbuf.

The resulting value of rbuf for every participating processing element is equal to the sum of all ranks i, where i ranges from 0 to numprocs – 1. If the value of rbuf for a participating processing element is not equal to this sum, an error is printed.

6.2 Parallel Virtual Machine

In the programs for testing the collective communication functions of the PVM library, no extra groups are defined. The set of all processing elements drawn on by the `mpprun` is the predefined global group. In all test programs, this group is referred to by the null name 0. Because no extra groups are defined, the rank of a processing element in the global group is equal to its processing element number. Note that the wrappers for the collective communication functions of the PVM library are constructed to support the use of different groups.

The test programs that are available for the collective communication functions of the PVM library all record the expected event trace if analysed with PAT. To record the pvm_bcast and pvm_mcast functions, all functions starting with ‘pvm_upk’ are intrumented too. The recorded number of bytes for an event and the resulting graphical visualisation of the logical pattern also meet the expectations. Therefore, it can be concluded that the wrappers for the collective communication functions of the PVM library are implemented correctly.

1-to-N (Collective) Communication Functions

☞ pvm_bcast

In the test program for the pvm_bcast function, the variable id stores the processing element number of a processing element. The value of numprocs represents the total number of processing elements in the global group. The processing element number of the root processing element is saved in the variable master. For all participating processing elements the tag value is initialised to 0.

The root processing element initialises a communication buffer for the integer buf, which is set to 255, with the pvm_initsend and pvm_pkint functions. The integer buf is sent to all other participating processing element using the pvm_bcast function. The receiving processing elements consult the active communication buffer to receive the data stored in the communication buffer. The pvm_upkint stores the received data into the variable buf. If the variable buf equals 255 for all participating processing elements, no error is generated.

☞ pvm_mcast

The root processing element of which the processing element number is saved in the variable master, multicasts the integer buf, set to 0, to all processing elements of which the processing element number is stored in the array list. All processing elements with an odd processing element number are selected to receive the data multicasted by the root processing element. Because the processing element number of the root processing element must not be in the array list, the value of the variable master equals 0.

Note that buf for all processing elements, including the root processing element, is initialised to 0. But for all processing elements with an odd processing element number buf is set to 1. This enables testing the correct functionality of the pvm_mcast function. The number of processing elements with an odd processing element number is equal to numprocs divided by two, for any value of numprocs.

After initialising the communication buffer by the root processing element, all processing elements with an odd processing element number receive the data in the communication buffer into the variable buf. If all processing elements have 0 stored in buf, no error has occurred.

☞ pvm_scatter

In the test program for the pvm_scatter function, the root processing element is the processing element with the largest processing element number, which is stored in master. The variable numprocs represents the number of all participating processing elements in the global group. To test the pvm_scatter function, the root processing element is selected for initialising an array for the data elements to be scattered. The value of sbuf[i], where sbuf is the name of the send array, is set to i, for all i ranging from 0 to numprocs – 1.

After calling the pvm_scatter function, every participating processing element stores the received data in the integer rbuf. The value of rbuf for a processing element must be equal to its processing element number. If this is true for all participating processing elements, no error is generated.

N-**to-1 Collective Communication Functions**

☞ pvm_gather

The integer value a participating processing element stores in the send buffer sbuf equals the result of (id + numprocs – 1)%numprocs. The variable id represents the processing element number and numprocs the total number of processing elements in the global group. To test the correct functionality of the pvm_gather function, the receive buffer rbuf of the root processing element is initialised with rbuf[i] = i, for all i ranging from 0 to numprocs – 1.

The effect of calling the pvm_gather function is that the value stored at rbuf[i] equals i – 1, for all i ranging from 1 to numprocs – 1 and that rbuf[0] = numprocs – 1. If this is true, the root processing element generates no gather error.

☞ pvm_reduce

The number of processing elements in the global group is saved in the variable numprocs. The variable id indicates the processing element number. Every participating processing element in the call of the pvm_reduce function stores the result of (id + numprocs – 1)%numprocs in the buffer rbuf. The addition function PvmSum is used in the call of the pvm_reduce function to calculate the sum of all values stored in the buffer rbuf by all participating processing elements.

The root processing element stores the result of the pvm_reduce function in the variable rbuf. After executing the pvm_reduce function the value of rbuf for the root processing element must equal the sum of the processing element numbers of all processing elements in the global group. A reduce error is generated if the value of rbuf for the root processing element is not changed to that result.

6.3 Shared Memory

This section discusses the programs used to test the SHMEM collective and atomic communication functions. In these test programs, the function _num_pes() of the SHMEM library is used to determine the total number of processing elements drawn on by the `mpprun` command. Instead of the standard **`malloc`**-function, the shmalloc function of the SHMEM library is used to allocate the memory space needed for several arrays. Using this function enables allocating the same address space in the memory of every processing element in the set of all participating processing elements. To ensure that all participating processing elements allocated memory space before calling a SHMEM collective communication function, the synchronisation function shmem_barrier_all() is called. The SHMEM function shfree frees memory space that is allocated with the shmalloc function.

Analysing the test programs that are available for the collective and atomic communication functions of the SHMEM library with PAT reveals that the expected event traces are recorded. Also the number of bytes recorded for an event and the resulting graphical visualisation of the recorded event traces meet the expectations. Therefore, it can be concluded that the wrappers for the collective and atomic communication functions of the SHMEM library are implemented correctly.

1-to-N Collective Communication Functions

☞ shmem_broadcast

To test the wrappers for the shmem_*broadcast* functions, a test program is written that uses the shmem_broadcast function. The array sbuf contains ten integer (**`long`**) values, ranging from 1 to 10. The variable nlong is used to indicate that sbuf contains ten elements. The number of processing elements drawn on by the `mpprun` command is saved in numprocs. After the initialisation of the synchronisation array pSync, the function shmem_barrier_all is used to ensure that all participating processing elements allocated memory space for the array pSync. The SHMEM library defines the variables _SHMEM_BCAST_SYNC_SIZE and _SHMEM_SYNC_VALUE.

Only the processing elements with an odd processing element number participate in the call of the shmem_broadcast function. The processing element number of the sending root processing element is saved in master. Note that the rank of the root processing element in the set of participating processing elements in the call of the shmem_broadcast function equals 0. All processing elements with an odd processing element number, except the root processing element, receive the broadcasted data elements of the array sbuf from the root processing element in the receive buffer rbuf.

After calling the shmem_broadcast function, the contents of the array rbuf must be equal to the contents of sbuf for the processing elements with an odd processing element number (except for the root processing element). If not, an error is generated.

N-to-M Collective Communication Functions with N-to-N default Pattern

☞ shmem_collect

The test program in which the shmem_collect function is called enables testing of the wrappers for the shmem_*collect* functions. The variable numprocs is used to store the number of processing elements drawn on by the `mpprun` command. The number of data elements that is sent by a processing element to all other participating processing elements equals its processing element number + 1 and is saved in nlong. The array sbuf stores integer values, ranging from 0 to nlong, which are sent to every other participating processing element.

The received integer values from every participating processing element are saved in the array rbuf after calling the shmem_collect function. The length of rbuf equals the sum of all values of nlong. However, before calling the shmem_collect function, a synchronising array pSync is initialised with _SHMEM_SYNC_VALUE. This variable and the length of pSync _SHMEM_COLLECT_SYNC_SIZE are defined in the SHMEM library.

Although not needed for the shmem_collect function, an array displ is initialised with the displacements from which the data elements in sbuf are sent to all other participating processing elements. Because displ is not used as an argument for the shmem_collect function, its memory space is allocated with the standard **`malloc`**-function. Note that the length of the array sbuf is different for every participating processing element. Therefore, the memory space for this array is allocated last.

No collect errors occur if all participating processing elements received i + 1 integers from the processing element with processing element number i, ranging from 0 to i and stored from position displ[i] in the array rbuf. The value of i ranges from 0 to numprocs - 1. If this is found to be true for every participating processing elements, no error is printed.

N-to-M Collective Communication Functions with N-to-1-to-N default Pattern

☞ shmem_fcollect

Testing the wrappers for the shmem_*fcollect* functions is done with the test program that calls the shmem_fcollect function. The variable numprocs is used to save the number of processing elements drawn on by the `mpprun` command. The array sbuf, of which the elements are sent to all other participating processing elements, stores numprocs integer values ranging from 0 to numprocs - 1. The array in which the results of the call of the shmem_fcollect function are stored, rbuf, is initialised with zeroes. The synchronising array pSync is of size _SHMEM_COLLECT_SYNC_SIZE and contains values equal to the value of the variable _SHMEM_SYNC_VALUE. These variables are both defined in the SHMEM library.

All processing elements with an odd processing element number participate in the call of the shmem_fcollect function. If the array rbuf of these processing elements contains numprocs integer values ranging from 0 to numprocs - 1, no error has occurred.

☞ shmem_int_sum_to_all

To test the wrappers for the shmem_*type_func*_to_all functions, a test program is available that calls the shmem_int_sum_to_all function. The nlong integer values stored in sbuf, ranging from 0 to nlong - 1, are sent to all other participating processing elements. After calling the shmem_int_sum_to_all function, the sum of every value stored at sbuf[i] in all participating processing elements is saved at rbuf[i]. The array rbuf is initialised with zeroes.

The synchronising array pSync, of size _SHMEM_REDUCE_SYNC_SIZE, is initialised with value _SHMEM_SYNC_VALUE. The size of the memory space allocated for the work array pWrk equals _SHMEM_REDUCE_MIN_WRKDATA_SIZE multiplied by the number of bytes needed to store an integer. The values of _SHMEM_REDUCE_SYNC_SIZE, _SHMEM_SYNC_VALUE and _SHMEM_REDUCE_MIN_WRKDATA_SIZE are initialised in the SHMEM library.

The processing elements of which the processing element number is odd participate in the call of the shmem_int_sum_to_all function. Therefore, the value stored at rbuf[i] must be equal to the multiplication of numprocs/2 with i. If this is not true for any of the processing elements with an odd processing element number, an error is generated.

Conditional Atomic Communication Functions

☞ shmem_int_cswap

The wrappers for the shmem_*type_cfunc* functions are tested with a test program that calls the shmem_int_cswap function. The call of the shmem_int_cswap function results in recording the pattern with a true condition under which data is exchanged. A program that calls the shmem_short_cswap function with a false condition is also available.

The processing element number of the processing element that calls the shmem_int_cswap function is saved in master. The processing element number of the remote processing element is represented by the variable remote. The integer target of the remote processing element that is inspected, is initialised with 13 for the remote processing element and with 0 for the calling processing element. The calling processing element stores the condition under which the data exchange occurs in the variable cond and is initialised to 13. So a data exchange takes place. The contents of the variable value, 7, is saved in to the variable target of the remote processing element after calling the shmem_int_cswap function.

If the return value of the shmem_int_cswap function, saved in the variable result, equals 13 and if the contents of the variable target of the remote processing element is 7, no errors are generated after calling the shmem_int_cswap function. Note that the shmem_barrier_all function is used to ensure that all variables are set to the correct values before and after calling the shmem_int_cswap function.

Unconditional Atomic Communication Functions

☞ shmem_int_swap

To test the wrappers for the shmem_*type_func* function, a test program is written that calls the shmem_int_swap function. The variable master stores the processing element number of the processing element that calls the shmem_int_swap function, whereas the variable remote stores the processing element number of the remote processing element. The contents of the variable value of the calling processing element is exchanged with the value saved in the variable target of the remote processing element. Before and after calling the shmem_int_swap function, the execution of the shmem_barrier_all function ensures that all variables are set correctly.

After calling the shmem_int_swap function, the variable result stores the value of the variable target of the remote processing element. Next to this, the value of the variable target of the remote processing element is changed to the contents of the variable value. If this is true, no errors have occurred.

7 Investigating the Collective Communication Functions

The wrappers for the collective communication functions of the message passing libraries are implemented without knowing the source code of the internal implementation of the collective communication functions for the Cray T3E. However, the set of wrappers for all communication functions of the message passing libraries enables investigating the internal implementation of the collective communication functions by analysing the test programs discussed in chapter 6 with PAT in a special way.

This chapter discusses how to investigate the internal implementations of the collective communication functions of the message passing libraries and the results of these investigations. After a special instrumentation of a test program, the program ‘piftest’ is used to investigate the recorded event trace. The program VAMPIR by Pallas is used for a graphical visualisation of the communication paths recorded during the execution of a test program. Note that the internal implementation of a collective communication function might depend on the number of participating processing elements and on the arguments specified in the call of the collective communication function.

7.1 Message Passing Interface

The collective communication functions of the MPI library are implemented using Point-to-Point communication functions of the SHMEM library only. To investigate the internal implementation of the MPI collective communication functions, a ‘`list`’ file is composed in which the names of all SHMEM and MPI communication functions are listed. This enables tracing of an MPI collective communication function together with its internally called SHMEM communication functions. Because the communication paths recorded for the internal implementation of several collective communication functions reveal to be similar, these are discussed simultaneously.

1-to-N Collective Communication Functions

☞ MPI_Bcast

The MPI_Bcast function is internally implemented using the shmem_put64 and shmem_put32 functions. First a kind of control part is simultaneously executed by every participating processing element in the call of the MPI_Bcast function. Every non-root processing element in the call of the MPI_Bcast function calls a shmem_put64 to send 64 bytes to the root processing element. A repeating structure, consisting of calls of the shmem_put32 function, is used to send data from the root processing element to all other participating processing elements. In this structure, every call of the shmem_put32 function is succeeded by a call of the shmem_put64 function for sending 8 bytes. Thereafter, a receiving processing element executes a shmem_put64 function for sending 8 bytes to itself. The structure is not continued until a receiving processing element called this shmem_put64 to itself.

If the number of participating processing elements including the root processing element, is a power of 2, the repeating structure seems to be more efficient.

☞ MPI_Scatter and MPI_Scatterv

The recorded event trace for the MPI_Scatter and MPI_Scatterv functions are similar. The MPI_Scatter function is probably seen as a special case of the MPI_Scatterv function.

The internal implementation of the MPI_Scatter and MPI_Scatterv functions consist of calls of the shmem_put64 and shmem_put32 functions. After the parallel execution of a kind of control part, consisting of shmem_put64 calls from every non-root processing element to the root processing element (64 bytes), the root processing element sequentially sends data to every non-root processing element using the shmem_put32 function succeeded by a shmem_put64 call. The root processing element also participates in the control process by executing a shmem_put64 to itself. Thereafter, every receiving processing element calls a shmem_put64 to itself of 8 bytes.

N-to-1 Collective Communication Functions

☞ MPI_Gather and MPI_Gatherv

Investigating the event traces recorded for the MPI_Gather and MPI_Gatherv functions reveals that the MPI_Gather function is probably seen as a special case of the MPI_Gatherv function. However, the execution of the MPI_Gather function takes rather much more time than executing the MPI_Gatherv function.

The shmem_put64 and shmem_put32 functions are used by the internal implementation of the MPI_Gather and MPI_Gatherv functions. The root processing element sequentially receives data from every other participating processing element due to a call of the shmem_put32 function after executing a shmem_put64 function. If a sending processing element leaves the MPI_Gather or MPI_Gatherv function, it calls a shmem_put64 function for sending 8 bytes to the root processing element. After receiving data from all participating processing elements other than the root processing element, the root processing element calls the shmem_put64 function to send 8 bytes to itself. The number of these final calls of the shmem_put64 function equals the total number of participating processing elements in the call of the MPI_Gather or MPI_Gatherv function.

☞ MPI_Reduce

The internal implementation of the MPI_Reduce function uses the shmem_put64 and shmem_put32 functions. Between the processing elements with an even processing element number, a kind of control part is executed. This is also done for the processing elements with an odd processing element number. Meanwhile, data is exchanged between the processing elements with even and odd processing element numbers.

Thereafter, a tree-structure of communication paths is recorded to send the results of the calculations, made because of the associative function used for the MPI_Reduce function, to the root processing element. Calling the shmem_put64 function and successively the shmem_put32 function implements this tree-structure. After every received result, the function shmem_put64 is called by the root processing element to send 8 bytes to itself.

The tree-structure seems to be more efficient if the number of participating processing elements in the call of the MPI_Reduce function is a power of 2.

N-to-M Collective Communication Functions

☞ MPI_Allgather, MPI_Allgatherv, MPI_Alltoall and MPI_Alltoallv

The communication paths recorded for the MPI_Allgather, MPI_Allgatherv, MPI_Alltoall and MPI_Alltoallv functions are rather similar. The MPI_Allgather, MPI_Allgatherv, MPI_Alltoall functions are probably seen as a special case of the MPI_Alltoallv function.

First, every participating processing element sends 8 bytes to itself, with a shmem_put64 call. Thereafter, every processing element participates in a kind of double tree-structure or "rotating" tree-structure in the sense of sending and receiving data. So, sending and receiving data is implemented with a tree-structure of calls of the shmem_put64 and shmem_put32 functions. Every call of the shmem_put32 function is preceded by a call of the shmem_put64 function. After participating in the execution of the double tree-structure, every participating processing element uses the shmem_put64 function to send 8 bytes to itself. The number of calls of these finalising shmem_put64 functions is equal to the total number of participating processing elements in the call of the concerning MPI collective communication function.

☞ MPI_Scan

The shmem_put64 and shmem_put32 functions are used to implement the MPI_Scan function. The interpretation of the recorded communication paths for the internal implementation of the MPI_Scan function is rather unclear. A kind of control part is succeeded by a tree-structure in which the processing elements with a higher rank number in the participating set of processing elements receive more data than the processing elements with lower rank numbers.

☞ MPI_Allreduce

The event trace recorded for the internal implementation of the MPI_Allreduce function is rather different if the number of participating processing elements is a power of 2 compared to the case in which this is not true. Both implementations use the shmem_put64 and shmem_put32 functions only.

The communication paths recorded for the internal implementation of the MPI_Allreduce function in the case that the number of processing elements is a power of 2, consists of two parts. The fist part is similar to the communication paths recorded for the MPI_Reduce function. However, the amount of exchanged data is more. In the second part, a tree-structure pattern of communication paths is recorded starting from one processing element to all other participating processing elements in order to send the results of the calculations made by the associative function used in the call of the MPI_Allreduce function. This tree-structure successively consists of calls of the shmem_put32, shmem_put64 and shmem_put32 functions. Thereafter, every participating processing element calls the shmem_put64 function to send 8 bytes to itself.

If the number of participating processing elements is a power of 2, a more efficient implementation is used. The recorded communication paths for this implementation consist of data exchanges in a binary tree-structure, implemented with shmem_put64 and shmem_put32 function calls. After receiving data, every participating processing element executes a shmem_put64 function to send 8 bytes to itself.

☞ MPI_Reduce_scatter

The internal implementation of the MPI_Reduce_scatter function uses the shmem_put64 and shmem_put32 functions. The communication pattern recorded for the internal implementation of the MPI_Reduce_scatter function consists of two parts. The first part is similar to the communication pattern recorded for the MPI_Reduce function. Thereafter, the recorded event trace is similar to the communication pattern recorded for the internal implementation of the MPI_Scatter and MPI_Scatterv functions. The internal implementation of the MPI_Reduce_scatter probably consists of calling the implementation used for the MPI_Reduce function succeeded by the implementation used for the MPI_Scatter and MPI_Scatterv functions.

7.2 Parallel Virtual Machine

Several SHMEM communication functions are used to implement the PVM collective communication functions. Next to this, the pvm_scatter and pvm_reduce functions call the pvm_psend and pvm_precv functions internally. To investigate the internal implementation of the PVM collective communication functions, a '*list*' file is composed in which the names of all SHMEM and PVM communication functions are listed. Recording the internally called pvm_psend and pvm_precv function is enabled by not checking the global variable internal_event_trace_gate in their wrappers. Note that pvm_psend and pvm_precv are implemented using a SHMEM Point-to-Point communication function.

1-to-N (Collective) Communication Functions

☞ pvm_bcast (with pvm_upkint)

The recorded event trace for the test program of the pvm_bcast function consists of calls of the shmem_int_swap and shmem_put64 functions. A tree-structure is used to implement the pvm_bcast function, starting with a call of the shmem_int_swap function. Thereafter, every remote processing element for the calls of these shmem_int_swap functions execute the shmem_int_swap function to exchange data in its own memory space. The number of bytes exchanged equals 8. Twice executing a sequence of the shmem_put64 and shmem_int_swap functions completes the tree-structure.

☞ pvm_mcast (with pvm_upkint)

The pvm_mcast function is implemented using a sequential structure of calls of the shmem_put64 and shmem_int_swap functions. Although the recorded communication paths looks similar to the communication paths recorded for the pvm_bcast functions, the structure of the communication paths recorded for the pvm_mcast function is sequential and not a parallel tree-structure.

☞ pvm_scatter

The implementation of the pvm_scatter function is similar to the implementation of the pvm_mcast function. But, the pvm_psend and pvm_precv functions (implemented with the shmem_long_p function) are also used by the internal implementation for exchanging data.

Next to this, an extra call of the shmem_int_swap function and three calls of the shmem_put64 function precede the recorded sequential structure to exchange data. After executing the sequential structure, an extra shmem_int_swap function is executed to exchange data in a participating processing element's own memory.

N-to-1 Collective Communication Functions

☞ pvm_reduce

The pvm_reduce function is implemented using the shmem_put64, shmem_int_swap and shmem_long_p functions for a tree-structure to exchange data. The shmem_long_p function is called because the internal implementation of the pvm_reduce function executes the pvm_psend and pvm_precv functions.

The recorded tree-structure reveals that the processing element with the lowest processing element number is assumed to be the root processing element. However, because this is not true, after the recorded tree-structure, the results are transferred to the real root processing element.

☞ pvm_gather

The implementation of the pvm_gather function consists of calls of the shmem_get64 and shmem_put64 functions. After a first communication part, implemented with calls of the shmem_get64 function, every processing element, including the root processing element, finally receives data via a shmem_put64 function.

7.3 Shared Memory

The SHMEM collective communication functions are implemented using SHMEM Point-to-Point communication functions. The '*list*' file composed for investigating the internal implementation of the SHMEM collective communication functions contains the names of all SHMEM communication functions. However, only if the environment variable PAT_TRACE_COLLECTIVE is not set, the calls of the Point-to-Point communication functions in the internal implementation of the SHMEM collective communication functions are recorded.

Note that the SHMEM atomic functions have an own implementation, not consisting of calls of any other communication function. The same is true for the Point-to-Point communication functions of the SHMEM library.

1-to-N Collective Communication Functions

☞ shmem_broadcast64

The internal implementation of the shmem_broadcast64 function uses the shmem_put64 and shmem_int_swap functions. The recorded event trace reveals a tree-structure, which is more efficient in the case that the number of participating processing element is a power of 2. The shmem_put64 function is used for a kind of control part before exchanging data with the shmem_int_swap function.

N-to-M Collective Communication Functions with N-to-N default pattern

☞ shmem_collect64

Implementing the wrappers for the shmem_*collect* functions revealed that the corresponding shmem_*fcollect* functions are called by their internal implementation.

To investigate when the shmem_fcollect64 function is executed by the internal implementation of the shmem_collect64 function, an additional event trace is recorded. Therefore, PAT_TRACE_COLLECTIVE is set and the function name shmem_collect64 is removed from the '`list`' file. Next to this, the wrapper for the shmem_fcollect64 function is changed so that the variable no_fcollect_event_trace_gate is not checked. The recorded event trace revealed that the shmem_fcollect64 function is called in the first stage of the internal implementation of the shmem_collect64 function.

Next to the shmem_fcollect64 function, the internal implementation also consists of calls of the shmem_int_swap and shmem_put64 functions. The communication pattern recorded after the shmem_fcollect64 function call consists of two tree-structures for exchanging data. If the number of participating processing elements is a power of 2, the tree-structures are more efficient.

N-to-M Collective Communication Functions with N-to-1-to-N default pattern

☞ shmem_fcollect64

The shmem_put64 and shmem_int_swap functions are used to implement the shmem_fcollect64 function. These functions are called to construct one tree-structure for receiving data. The internal implementation of the shmem_fcollect64 function is more efficient in the case that the number of participating processing elements in the call of the shmem_fcollect64 function is a power of 2.

☞ shmem_int_sum_to_all

The shmem_int_sum_to_all function is implemented using the shmem_put64 function only. A sequential structure is constructed with calls of the shmem_put64 function for sending data. Thereafter, data is received with another sequential structure. This finally results in the exchange of data from all processing elements to all other processing elements. According to [`man shmem_int_sum_to_all`], the execution of the shmem_int_sum_to_all function is more efficient if the number of participating processing elements is a power of 2.

8 Recommendations

Although the results, discussed in chapter 7, are rather hypothetical, some recommendations for the internal implementation of the collective communication functions can be given. However, first several general remarks are presented.

General

Instrumenting a parallel program is rather time-consuming. Next to this, the memory space needed to store the instrumented parallel program is very large compared to the memory space needed for storing the original parallel program. Furthermore, a large amount of the execution time of an instrumented parallel program is caused by the large number of function calls that is made during the execution of the instrumented parallel program.

Including the possibility for tracing communication functions in the internal implementation of the communication functions might speed up the execution time of a parallel program considerable. No instrumentation process would be necessary to gain an event trace of the occurred communication paths. An environment variable could be used to switch tracing communication events on and off.

The arguments of many collective communication functions of the MPI library specify the type of the sent and received data. However, the type of data handled by a collective communication function is not changed during the execution of the collective communication function. If the arguments for the type of the sent and received data are not registered in the MPI standard, it is not clear why both the types of the sent and received data must be specified.

The way in which a set of participating processing elements has to be specified for SHMEM collective communication functions is rather restrictive. Probably the internal implementation of the SHMEM collective communication functions uses the representation for efficiency reasons. Note that if the number of participating processing elements is a power of 2, the execution of a SHMEM collective communication function is more efficient.

Analysing a parallel program in the way discussed in chapter 7 enables checking whether the parallel program is correct. If extremely much communication paths are recorded for a collective communication function, the executed parallel program may not be correct. An extreme large number of recorded communication paths could be caused by incorrect sizes of the send and/or receive buffers containing the data elements to be sent or received respectively.

Internal Implementations

The investigations of chapter 7 showed that the internal implementation of many collective communication functions use shmem_put64 and shmem_int_swap functions to exchange data in the memory of a single processing element. This probably denotes on the existence of a special memory for communication purposes. Using this special memory might increase the execution speed of a collective communication function.

The internal implementations of the collective communication functions of the message passing libraries consist of SHMEM Point-to-Point communication function calls. However, the internal implementation of some PVM and SHMEM collective communication functions also uses other communication functions of their own library. Because the internally called communication functions are also implemented with SHMEM Point-to-Point communication functions, this results in inefficiency. Using the indirectly called SHMEM Point-to-Point communication functions in the internal implementation of the collective communication functions directly would increase execution efficiency.

Although the investigation of the internal implementation of the MPI_Gather function showed that the MPI_Gather function is probably a special case of the MPI_Gatherv function, it is not clear why the execution of the MPI_Gather function takes much more time than the execution of the MPI_Gatherv function.

The investigation of the internal implementation of the MPI_Allgather function revealed that the MPI_Allgather function is probably seen as a special case of the MPI_Alltoallv function. Therefore, the wrapper of the MPI_Allgather should probably also record the N-to-N logical pattern by default instead of the N-to-1-to-N logical pattern.

9 Conclusions

The wrappers developed for the communication functions of the message passing libraries consist of an entrance and exit hook. The general frameworks for the wrappers of the different forms of collective communication reveal that recording logical communication patterns is relatively uncomplicated. However, the final implementation of the wrappers for the collective communication functions of the message passing libraries available on the Cray T3E parallel computer system requires adaptations of the general frameworks.

To implement the wrappers for the collective communication functions of the MPI library, only minor adaptations of the general frameworks are necessary. The implementation of the wrappers for the collective communication functions of the PVM library involves more adaptations, whereas implementation of the SHMEM collective communication functions requires rather major adaptations of the general frameworks. The wrappers for the atomic communication functions of the SHMEM library are very different to the wrappers for the collective communication functions.

According to the performed tests, the wrappers for the collective and atomic communication functions of the message passing libraries are implemented correctly. Although the test programs do not test the communication functions in all possible situations, the wrappers record the correct logical communication pattern by supporting all options available in a message passing library.

Investigating the internal implementation of the collective communication functions of the message passing libraries reveals that several collective communication functions could be implemented more efficiently. The internal implementation of the collective communication functions mainly consists of calls of SHMEM Point-to-Point communication functions. However, some collective communication functions internally call other communication functions of the same library, which are implemented with SHMEM Point-to-Point communication functions again.

By including the wrappers for the collective and atomic communication functions in the ‘CrayTools’ module, the dynamic behaviour of all communicating processes in a parallel program can be analysed with PAT.

Acknowledgement

At the Eindhoven University of Technology, in the Netherlands, the curriculum for the study Information Technology Science includes an internship to be completed at a company. In extension of my previous projects, practical experience with complex parallel computer systems and their system software was a logical next stage.

Because the Zentralinstitut für Angewandte Mathematik at the Forschungszentrum Jülich is unique in the region concerning the deployment of complex parallel computer systems, I was very pleased that Prof. F. Hoßfeld and R. Esser granted my request for completing an internship at the Zentralinstitut für Angewandte Mathematik. The experiences that I gained during the development of solutions for the problem statement in this internship perfectly supplemented the knowledge obtained from the study Information Technology Science.

In this acknowledgement, I would like to thank everyone who contributed to the realisation of the results of my internship at the Zentralinstitut für Angewandte Mathematik. Special thanks go to my advisor B. Mohr, who was very helpful in answering many of my questions, even when he stayed in the U.S.A. in the final stage of my internship. I really enjoyed working together with B. Mohr.

Finally, I would like to thank everyone who made my stay at the Zentralinstitut für Angewandte Mathematik very pleasant. I wish everyone the best for their future studies and careers.

Appendix

This appendix enumerates the source code of programs used to test the wrappers for the communication functions of the message passing libraries discussed in this report. Note that the test programs published in this appendix are subjected to changes.

Message Passing Interface

1-to-N Collective Communication Functions

‘mpitestC1.c’ tests the MPI_Bcast function

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

int main(int argc, char *argv[]) {

  /* General initialisations            */

  int id, numprocs, master;
  int buf;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  master = numprocs - 1;

  if (numprocs == 1) {
    fprintf(stderr,"Using more than 1 CPU makes the tests of this program"
           " more interesting\n");
    exit(EXIT_FAILURE);
  }

  /* Broadcast function                 */

  if (id == master) buf = 255;

  MPI_Bcast(&buf, 1, MPI_INT, master, MPI_COMM_WORLD);

  if (buf != 255)
    fprintf(stderr, "CPU: %d Broadcast Error: got %d\n", id, buf);

  /* Finalises                          */

  MPI_Finalize();
  exit(EXIT_SUCCESS);
}
```

‘mpitestC0.c’ tests the MPI_Scatter function

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

int main(int argc, char *argv[]) {
```

```
  /* General initialisations            */

  int id, numprocs, master;
  int rbuf;
  int *sbuf = NULL;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  master = numprocs - 1;

  if (numprocs == 1) {
    fprintf(stderr,"Using more than 1 CPU makes the tests of this program"
           " more interesting\n");
    exit(EXIT_FAILURE);
  }

  /* Filling initial array in master CPU*/

  if (id == master) {
    int i;

    rbuf = 17;
    sbuf = malloc( sizeof(int) * numprocs );
    for (i = 0; i < numprocs; i++)
      sbuf[i] = (i +25) ;
  }

  /* Scatter function                   */

  MPI_Scatter(sbuf, 1, MPI_INT, &rbuf, 1, MPI_INT, master,
              MPI_COMM_WORLD);

  if ( rbuf != (id + 25) )
    fprintf(stderr, "CPU: %d Scatter Error: got %d\n", id, rbuf);

  /* Finalises                          */

  if (id == master) free(sbuf);
  MPI_Finalize();
  exit(EXIT_SUCCESS);
}
```

‘mpitestC8.c’ tests the MPI_Scatterv function

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

int main(int argc, char *argv[]) {

  /* General initialisations            */

  int id, numprocs, master, ii;
  int *rbuf = NULL;
  int *sbuf = NULL, *displ = NULL, *scounts = NULL;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
```

```
  master = numprocs - 1;

  if (numprocs == 1) {
    fprintf(stderr,"Using more than 1 CPU makes the tests of this program"
           " more interesting\n");
    exit(EXIT_FAILURE);
  }

  /* Filling initial array in master CPU*/

  if (id == master) {
    int i, j, d, len;            /* i, j and d are counters           */

    /* Note: if scounts[i] = 0, no data is sent to CPU: i !! */

    scounts = malloc( sizeof(int) * numprocs);
    displ = malloc( sizeof(int) * numprocs);
    len = 0;                     /* scounts and displ initialisations  */
    for (i = 0; i < numprocs; i++) {
      scounts[i] = i + 1;        /* number of elements sent CPU: i     */
      displ[i] = len;            /* displacements in sbuf for CPU: i   */
      len = len + i + 1;
    }
    sbuf = malloc( sizeof(int) * len );
    j = d = 0;
    while (j < numprocs) {       /* fill send array: for every CPU     */
                                 /* there are his id+1 elements,       */
      for (i = 0; i <= j; i++) { /* containing 0 to id                 */
        sbuf[d + i] = i;
      }
      d = displ[i];
      j++;
    }
  }
  rbuf = malloc( sizeof(int) * (id + 1));

  /* Scatterv function                  */

  MPI_Scatterv(sbuf, scounts, displ, MPI_INT,
               rbuf, id + 1, MPI_INT, master, MPI_COMM_WORLD);

  for (ii = 0; ii < (id + 1); ii++)
    if (rbuf[ii] != ii)
      fprintf(stderr, "CPU: %d Scatterv Error: got %d\n", id, rbuf[ii]);

  /* Finalises                          */

  if (id == master) {
    free(sbuf);
    free(scounts);
    free(displ);
  }

  free(rbuf);
  MPI_Finalize();
  exit(EXIT_SUCCESS);
}
```

N-to-1 Collective Communication Functions

‘mpitestC2.c’ tests the MPI_Gather function

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

int main(int argc, char *argv[]) {

  /* General initialisations            */

  int id, numprocs, master;
  int sbuf;
  int *rbuf = NULL;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  master = numprocs - 1;

  if (numprocs == 1) {
    fprintf(stderr,"Using more than 1 CPU makes the tests of this program"
           " more interesting\n");
    exit(EXIT_FAILURE);
  }

  /* Initialise receiving array of master  */

  if (id == master)
    rbuf = malloc( sizeof(int) * numprocs );

  /* Gather function                    */

  MPI_Gather(&id, 1, MPI_INT,
             rbuf, 1, MPI_INT, master, MPI_COMM_WORLD);

  if (id == master) {
    int i;

    for (i = 0; i < numprocs; i++) {
      if (rbuf[i] != i)
        fprintf(stderr, "CPU: %d Gather Error: got %d\n", i, rbuf[i]);
    }
    free(rbuf);
  }

  /* Finalises                          */

  MPI_Finalize();
  exit(EXIT_SUCCESS);
}
```

‘mpitestC9.c’ tests the MPI_Gatherv function

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
```

```
int main(int argc, char *argv[]) {

  /* General initialisations            */

  int id, numprocs, master, ii;
  int *sbuf = NULL;
  int *rbuf = NULL, *displ = NULL, *rcounts = NULL;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  master = numprocs - 1;

  if (numprocs == 1) {
    fprintf(stderr,"Using more than 1 CPU makes the tests of this program"
           " more interesting\n");
    exit(EXIT_FAILURE);
  }

  /* Filling sending array              */

  sbuf = malloc( sizeof(int) * (id + 1) );
  for (ii = 0; ii < (id + 1); ii++) sbuf[ii] = ii;

  /* Filling initial array's in master CPU */

  if (id == master) {
    int i, j, d, len;            /* i, j and d are counters           */

    /* Note: if rcounts[i] = 0, no data is received from CPU: i !! */

    rcounts = malloc( sizeof(int) * numprocs);
    displ = malloc( sizeof(int) * numprocs);
    len = 0;                     /* scounts and displ initialisations  */
    for (i = 0; i < numprocs; i++) {
      rcounts[i] = i + 1;        /* number of elements sent CPU: i     */
      displ[i] = len;            /* displacements in rbuf for CPU: i   */
      len = len + i + 1;
    }
    rbuf = malloc( sizeof(int) * len );
  }

  /* Gatherv function                   */

  MPI_Gatherv(sbuf, id + 1, MPI_INT, rbuf, rcounts,
              displ, MPI_INT, master, MPI_COMM_WORLD);

  if (id == master) {
    int i, j, d;                 /* i, j and d are counters           */

    j = d = 0;
    while (j < numprocs) {       /* check recv array: for every CPU    */
                                 /* there should be his id+1 elements  */
      for (i = 0; i <= j; i++) { /* containing 0 to his id             */
        if (rbuf[d + i] != i)
          fprintf(stderr, "CPU: %d Gatherv Error: got %d from CPU: %d\n",
                  id, rbuf[d + i], j);
      }
      d = displ[i];
      j++;
    }
```

```
  }

  /* Finalises                          */

  if (id == master) {
    free(rbuf);
    free(rcounts);
    free(displ);
  }
  free(sbuf);
  MPI_Finalize();
  exit(EXIT_SUCCESS);
}
```

‘mpitestC3.c’ tests the MPI_Reduce function

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

int main(int argc, char *argv[]) {

  /* General initialisations            */

  int numprocs, master, id, i;
  int *sbuf = NULL, *rbuf = NULL;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  master = numprocs - 1;

  if (numprocs == 1) {
    fprintf(stderr,"Using more than 1 CPU makes the tests of this program"
           " more interesting\n");
    exit(EXIT_FAILURE);
  }

  /* Initialising buffer array's        */

  sbuf = malloc( sizeof(int) * numprocs );
  for (i = 0; i < numprocs; i++) sbuf[i] = id;
  if (id == master) rbuf = malloc( sizeof(int) * numprocs );

  /* Reduce function                    */

  /* Note: the length of rbuf MUST be equal to the length of sbuf !! */

  MPI_Reduce(sbuf, rbuf, numprocs, MPI_INT, MPI_SUM, master,
             MPI_COMM_WORLD);

  if (id == master) {
    int i, temp;

    temp = 0;
    for (i = 0; i < numprocs; i++) temp = temp + i;

    for (i = 0; i < numprocs; i++)
      if (rbuf[i] != temp)
```

```
        fprintf(stderr, "CPU: %d Reduce Error: got %d instead of %d\n",
                master, rbuf[i], temp);
  }

  /* Finalises                          */

  if (id == master) free(rbuf);
  free(sbuf);
  MPI_Finalize();
  exit(EXIT_SUCCESS);
}
```

N-to-M Collective Communication Functions with N-to-N (default) Pattern

'mpitestC10.c' tests the MPI_Allgatherv function

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

int main(int argc, char *argv[]) {

  /* General initialisations            */

  int id, numprocs, ii;
  int *sbuf = NULL;
  int *rbuf = NULL, *displ = NULL, *rcounts = NULL;
  int i, j, d, len;                 /* i, j and d are counters         */

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);

  if (numprocs == 1) {
    fprintf(stderr,"Using more than 1 CPU makes the tests of this program"
           " more interesting\n");
    exit(EXIT_FAILURE);
  }

  /* Filling sending array              */

  sbuf = malloc( sizeof(int) * (id + 1) );
  for (ii = 0; ii < (id + 1); ii++) sbuf[ii] = ii;

  /* Note: if rcounts[i] = 0, no data is received from CPU: i !! */

  rcounts = malloc( sizeof(int) * numprocs);
  displ = malloc( sizeof(int) * numprocs);
  len = 0;                           /* scounts and displ initialisations  */
  for (i = 0; i < numprocs; i++) {
    rcounts[i] = i + 1;              /* number of elements sent CPU: i     */
    displ[i] = len;                  /* displacements in rbuf for CPU: i   */
    len = len + i + 1;
  }
  rbuf = malloc( sizeof(int) * len );

  /* Allgatherv function                */

  MPI_Allgatherv(sbuf, id + 1, MPI_INT, rbuf, rcounts,
                 displ, MPI_INT, MPI_COMM_WORLD);
```

```
  j = d = 0;
  while (j < numprocs) {         /* check recv array: for every CPU    */
                                 /* there should be his id+1 elements  */
    for (i = 0; i <= j; i++) {   /* containing 0 to his id             */
      if (rbuf[d + i] != i)
        fprintf(stderr, "CPU: %d Allgatherv Error: got %d from CPU: %d\n",
                id, rbuf[d + i], j);
    }
    d = displ[i];
    j++;
  }

  /* Finalises                           */

  free(sbuf);
  free(rbuf);
  free(rcounts);
  free(displ);
  MPI_Finalize();
  exit(EXIT_SUCCESS);
}
```

‘mpitestC4.c’ tests the MPI_Alltoall function

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

int main(int argc, char *argv[]) {

  /* General initialisations            */

  int id, numprocs;
  int *sbuf = NULL, *rbuf = NULL;
  int count;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  master = numprocs - 1;

  if (numprocs == 1) {
    fprintf(stderr,"Using more than 1 CPU makes the tests of this program"
           " more interesting\n");
    exit(EXIT_FAILURE);
  }

  /* Buffer initialisations             */

  rbuf = malloc( sizeof(int) * numprocs );
  sbuf = malloc( sizeof(int) * numprocs );
  for (count = 0; count < numprocs; count++)
    sbuf[count] = count;

  /* All to All instruction             */

  MPI_Alltoall(sbuf, 1, MPI_INT, rbuf, 1, MPI_INT, MPI_COMM_WORLD);

  for (count = 0; count < numprocs; count++) {
    if (rbuf[count] != id)
```

```
      fprintf(stderr, "CPU: %d All to All Error: got %d of CPU %d\n",
              id, rbuf[count], count);
  }

  /* Finalises                          */

  free(rbuf);
  free(sbuf);
  MPI_Finalize();
  exit(EXIT_SUCCESS);
}
```

‘mpitestC11.c’ tests the MPI_Alltoallv function

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

int main(int argc, char *argv[]) {

  /* General initialisations            */

  int id, numprocs;
  int *sbuf = NULL, *scounts = NULL, *sdispl = NULL;
  int *rbuf = NULL, *rcounts = NULL, *rdispl = NULL;
  int i, j, d, len;       /* i, j and d are counters     */

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);

  if (numprocs == 1) {
    fprintf(stderr,"Using more than 1 CPU makes the tests of this program"
           " more interesting\n");
    exit(EXIT_FAILURE);
  }

  /* Initialising sending control array's  */

  scounts = malloc( sizeof(int) * numprocs );
  sdispl = malloc( sizeof(int) * numprocs );

  /* Note: if scounts[i] = 0, no data is sent to CPU: i !! */

  len = 0;                        /* scounts and sdispl init's          */
  for (i = 0; i < numprocs; i++) {
    scounts[i] = i + 1;           /* number of elements sent to CPU: i  */
    sdispl[i] = len;              /* displacements in sbuf for CPU: i */
    len = len + i + 1;
  }

  /* Initialising sending data array    */

  sbuf = malloc( sizeof(int) * len );
  j = d = 0;
  while (j < numprocs) {
    for (i = 0; i <= j; i++) {
      sbuf[d + i] = i;
    }
    d = sdispl[i];
```

```
    j++;
  }

  /* Initialising receiving control array's  */

  rcounts = malloc( sizeof(int) * numprocs );
  rdispl = malloc( sizeof(int) * numprocs );

  /* Note: if rcounts[i] = 0, no data is received from CPU: i !! */

  for (i = 0; i < numprocs; i++) {      /* rcounts and rdispl init's   */
    rcounts[i] = id + 1;         /* number of elements recv for CPU: i */
    rdispl[i] = i * (id + 1);    /* displacements in rbuf for CPU: i   */
  }

  /* Initialising receiving data array  */

  rbuf = malloc( sizeof(int) * numprocs * (id + 1) );
  for (i = 0; i < ( numprocs * (id + 1) ); i++) rbuf[i] = 0;

  /* Alltoallv function                 */

  MPI_Alltoallv(sbuf, scounts, sdispl, MPI_INT, rbuf, rcounts, rdispl,
                MPI_INT, MPI_COMM_WORLD);

  j = d = 0;
  while (j < numprocs) {
    for (i = 0; i < (id + 1); i++) {
      if (rbuf[d + i] != i)
        fprintf(stderr, "CPU: %d Alltoallv Error: got %d from CPU: %d\n",
                id, rbuf[d + i], j);
    }
    d = rdispl[i];
    j++;
  }

  /* Finalises                          */

  free(sbuf);
  free(rbuf);
  free(scounts);
  free(rcounts);
  free(sdispl);
  free(rdispl);
  MPI_Finalize();
  exit(EXIT_SUCCESS);
}
```

‘mpitestC14.c’ tests the MPI_Scan function

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

int main(int argc, char *argv[]) {

  /* General initialisations            */

  int id, numprocs, i, temp;
  int *sbuf = NULL, *rbuf = NULL;
```

```
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);

  if (numprocs == 1) {
    fprintf(stderr,"Using more than 1 CPU makes the tests of this program"
           " more interesting\n");
    exit(EXIT_FAILURE);
  }

  /* Initialisation of array's          */

  sbuf = malloc( sizeof(int) * numprocs );
  for (i = 0; i < numprocs; i++) sbuf [i] = id;
  rbuf = malloc( sizeof(int) * numprocs );
  for (i = 0; i < numprocs; i++) rbuf[i] = 0;

  /* Note: the length of sbuf MUST be equal to the length of rbuf */

  /* Scan function                      */

  MPI_Scan(sbuf, rbuf, numprocs, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

  temp = 0;
  for (i = 0; i < id + 1; i++) temp = temp + i;

  for (i = 0; i < numprocs; i++) {
    if (rbuf[i] != temp )
      fprintf(stderr, "CPU: %d Scan Error: got %d instead of %d\n",
              id, rbuf[i], temp);
  }

  /* Finalises                          */

  free(sbuf);
  free(rbuf);
  MPI_Finalize();
  exit(EXIT_SUCCESS);
}
```

N-to-M Collective Communication Functions with N-to-1-to-N default Pattern

‘mpitestC7.c’ tests the MPI_Allgather function

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

int main(int argc, char *argv[]) {

  /* General initialisations            */

  int id, numprocs, turn;
  int sbuf;
  int *rbuf = NULL;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
```

```
  if (numprocs == 1) {
    fprintf(stderr,"Using more than 1 CPU makes the tests of this program"
           " more interesting\n");
    exit(EXIT_FAILURE);
  }

  /* Initialise receiving array         */

  rbuf = malloc( sizeof(int) * numprocs );

  /* Allgather function                 */

  MPI_Allgather(&id, 1, MPI_INT, rbuf, 1, MPI_INT, MPI_COMM_WORLD);

  for (turn = 0; turn < numprocs; turn++) {
    if (id == turn)  {
      int i;

      for (i = 0; i < numprocs; i++) {
        if (rbuf[i] != i)
          fprintf(stderr, "CPU: %d Allgather Error: got %d of CPU: %d\n",
                  id, rbuf[i], i);
      }
      free(rbuf);
    }
  }

  /* Finalises                          */

  MPI_Finalize();
  exit(EXIT_SUCCESS);
}
```

‘mpitestC12.c’ tests the MPI_Allreduce function

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

int main(int argc, char *argv[]) {

  /* General initialisations            */

  int id, numprocs;
  int *rbuf = NULL, *sbuf = NULL;
  int i, temp;                        /* i and temp are counters       */

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);

  if (numprocs == 1) {
    fprintf(stderr,"Using more than 1 CPU makes the tests of this program"
           " more interesting\n");
    exit(EXIT_FAILURE);
  }

  /* Initialising buffer array's        */
```

```
  rbuf = malloc( sizeof(int) * numprocs );
  sbuf = malloc( sizeof(int) * numprocs );
  for (i = 0; i < numprocs; i++) sbuf[i] = id;

  /* Allreduce function                */

  /* Note: the length of rbuf MUST be equal to the length of sbuf !! */

  MPI_Allreduce(sbuf, rbuf, numprocs, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

  temp = 0;
  for (i = 0; i < numprocs; i++) temp = temp + i;
  for (i = 0; i < numprocs; i++)
   if (rbuf[i] != temp)
    fprintf(stderr, "CPU: %d Allreduce Error: got %d instead of %d\n",
            id, rbuf[i], temp);

  /* Finalises                         */

  free(rbuf);
  free(sbuf);
  MPI_Finalize();
  exit(EXIT_SUCCESS);
}
```

‘mpitestC13.c’ tests the MPI_Reduce_scatter function

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

int main(int argc, char *argv[]) {

  /* General initialisations           */

  int id, numprocs, i, temp;
  int *sbuf = NULL, rbuf, *rcounts = NULL;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);

  if (numprocs == 1) {
    fprintf(stderr,"Using more than 1 CPU makes the tests of this program"
           " more interesting\n");
    exit(EXIT_FAILURE);
  }

  /* Initialisation of array's         */

  sbuf = malloc( sizeof(int) * numprocs );
  for (i = 0; i < numprocs; i++) sbuf [i] = id;
  rcounts = malloc( sizeof(int) * numprocs );

  /* Note if rcounts[i] = 0 then CPU: i will not receive data !! */

  for (i = 0; i < numprocs; i++) rcounts[i] = 1;

/* Note: The length of sbuf MUST be equal to the sum of all rcounts[i] !!
     The length of CPU: i his rbuf MUST be equal to rcounts[i] */
```

```
  /* Reduce_scatter function            */

  MPI_Reduce_scatter(sbuf, &rbuf, rcounts, MPI_INT, MPI_SUM,
                     MPI_COMM_WORLD);

  temp = 0;
  for (i = 0; i < numprocs; i++) temp = temp + i;
  if (rbuf != temp )
    fprintf(stderr, "CPU: %d Reduce_scatter Error: got %d instead of %d\n",
            id, rbuf, temp);

  /* Finalises                          */

  free(sbuf);
  free(rcounts);
  MPI_Finalize();
  exit(EXIT_SUCCESS);
}
```

Parallel Virtual Machine

1-to-N (Collective) Communication Functions

'pvmtestC0.c' tests the pvm_bcast function

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include "pvm3.h"

int main(void) {
  int id, numprocs, master;
  int buf, tag;

  id = pvm_get_PE( pvm_mytid() );
  numprocs = pvm_gsize( 0 );
  master = numprocs - 1;
  tag = 0;

  if (numprocs == 1) {
    fprintf(stderr,"Using more than 1 CPU makes the tests in this program"
           " more interesting\n");
    exit(EXIT_FAILURE);
  }

  /* Broadcast function                 */

  if (id == master)  {
    buf = 255;
    pvm_initsend(PvmDataRaw);
    pvm_pkint(&buf, 1, 1);
    pvm_bcast(0, tag);
  }
  else  {
    pvm_recv(master, tag);
    pvm_upkint(&buf, 1, 1);
  }
```

```
  if (buf != 255)
    fprintf(stderr, "CPU: %d Broadcast Error: got %d\n", id, buf);

  /* Finalises                          */

  pvm_exit();
  exit(EXIT_SUCCESS);
}
```

‘pvmtestC1.c’ tests the pvm_mcast function

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include "pvm3.h"

int main(void) {
  int id, numprocs, master;
  int buf, tag;

  id = pvm_get_PE( pvm_mytid() );
  numprocs = pvm_gsize( 0 );
  master = 0;        /* master = 0 used for simplicity                */
                     /* master MUST be excluded from mcast's tid list  */
  tag = 0;

  if (numprocs == 1) {
    fprintf(stderr,"Using more than 1 CPU makes the tests in this program"
           " more interesting\n");
    exit(EXIT_FAILURE);
  }

  /* Multicast function                 */

   if ( id%2 == 0 )
    buf = 0;
  else
    buf = 1;

  if (id == master) {
    int i, count;
    int *list = NULL;

    list = malloc( sizeof(int) * (numprocs/2) );
    count = 0;
    for (i = 0; i < numprocs; i++)  {
      if (i%2 == 1) {
        list[count] = i;
        count++;
      }
    }

    pvm_initsend(PvmDataRaw);
    pvm_pkint(&buf, 1, 1);
    pvm_mcast(list, numprocs/2, tag);

    free(list);
  }
```

```
  if ( id%2 == 1) {
    pvm_recv(0, tag);
    pvm_upkint(&buf, 1, 1);
  }

  if (buf != 0)
    fprintf(stderr, "CPU: %d Multicast Error: got %d instead of 0\n",
            id, buf);

  /* Finalises                          */

  pvm_exit();
  exit(EXIT_SUCCESS);
}
```

‘pvmtestC2.c’ tests the pvm_scatter function

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include "pvm3.h"

int main(void) {
  int id, numprocs, master;
  int rbuf, tag;
  int *sbuf = NULL;

  id = pvm_get_PE( pvm_mytid() );
  numprocs = pvm_gsize( 0 );
  master = numprocs - 1;
  tag = 0;

  if (numprocs == 1) {
    fprintf(stderr,"Using more than 1 CPU makes the tests in this program"
           " more interesting\n");
    exit(EXIT_FAILURE);
  }

  /* Filling initial array in master CPU*/

  if ( id == master ) {
    int i;

    sbuf = malloc( sizeof(int) * numprocs );
    for (i = 0; i < numprocs; i++)
      sbuf[i] = i;
  }

  /* Scatter function                   */

  pvm_scatter(&rbuf, sbuf, 1, PVM_INT, tag, 0, master);

  if ( rbuf != id )
    fprintf(stderr, "CPU: %d Scatter Error: got %d\n", id, rbuf);

  /* Finalises                          */

  if (id == master)
    free(sbuf);
```

```
  pvm_exit();
  exit(EXIT_SUCCESS);
}
```

N-to-1 Collective Communication Functions

‘pvmtestC3.c’ tests the pvm_gather function

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include "pvm3.h"

int main(void) {
  int id, numprocs, master;
  int sbuf, tag;
  int *rbuf = NULL;

  id = pvm_get_PE( pvm_mytid() );
  numprocs = pvm_gsize( 0 );
  master = numprocs - 1;
  tag = 0;

  if (numprocs == 1) {
    fprintf(stderr,"Using more than 1 CPU makes the tests in this program"
           " more interesting\n");
    exit(EXIT_FAILURE);
  }

  /* Filling initial array in master CPU*/

  if ( id == master ) {
    int i;

    rbuf = malloc( sizeof(int) * numprocs );
    for (i = 0; i < numprocs; i++) rbuf[i] = i;
  }

  /* Gather function                    */

  sbuf = (id + numprocs - 1)%numprocs;
  pvm_gather(rbuf, &sbuf, 1, PVM_INT, tag, 0, master);

  if (id == master) {
    int i;

    if (rbuf[0] != (numprocs - 1))
      fprintf(stderr, "CPU: %d Gather Error: got %d\n", 0, rbuf[0]);
    for (i = 1; i < numprocs; i++) {
      if (rbuf[i] != (i - 1))
        fprintf(stderr, "CPU: %d Gather Error: got %d\n", i, rbuf[i]);
    }
    free(rbuf);
  }

  /* Finalises                          */

  pvm_exit();
  exit(EXIT_SUCCESS);
}
```

‘pvmtestC4.c’ tests the pvm_reduce function

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include "pvm3.h"

int main(void) {
  int id, numprocs, master;
  int rbuf, tag;

  id = pvm_get_PE( pvm_mytid() );
  numprocs = pvm_gsize( 0 );
  master = numprocs - 1;
  tag = 0;

  if (numprocs == 1) {
    fprintf(stderr,"Using more than 1 CPU makes the tests in this program"
           " more interesting\n");
    exit(EXIT_FAILURE);
  }

  /* Reduce function                    */

  rbuf = (id + numprocs - 1)%numprocs;
  pvm_reduce(PvmSum, &rbuf, 1, PVM_INT, tag, 0, master);

  if (id == master) {
    int i, temp;

    temp = 0;
    for (i = 0; i < numprocs; i++)
      temp = temp + i;
    if (rbuf != temp)
      fprintf(stderr, "Reduce Error: root CPU: %d got %d instead of %d\n",
              master, rbuf, temp);
  }

  /* Finalises                          */

  pvm_exit();
  exit(EXIT_SUCCESS);
}
```

Shared memory

1-to-N Collective Communication Functions

‘smatestC0.c’ tests the shmem_broadcast function

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <mpp/shmem.h>

int main(int argc, char *argv[]) {

  /* General initialisations            */

  int ii, numprocs, master, count;
```

```
  static long sbuf[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
  static long rbuf[10];
  int nlong;
  long *pSync = NULL;

  numprocs = _num_pes();
  master = 1;
  nlong = 10;

  if (numprocs == 1) {
    fprintf(stderr,"Using more than 1 CPU makes the tests of this program"
           " more interesting\n");
    exit(EXIT_FAILURE);
  }

  for (ii = 0; ii < nlong; ii++) rbuf[ii] = 0;
  pSync = shmalloc(sizeof(long) *_SHMEM_BCAST_SYNC_SIZE);
  for (ii=0; ii < _SHMEM_BCAST_SYNC_SIZE; ii++) {
    pSync[ii] = _SHMEM_SYNC_VALUE;
  }

  shmem_barrier_all();      /* Wait for all CPUs to initialise pSync   */

  /* Broadcast function                */

  if (_my_pe()%2 == 1)
    shmem_broadcast(rbuf, sbuf, nlong, 0, 1, 1,
                    numprocs/2, pSync);      /* local master CPU = 0   */

  if ((_my_pe()%2 == 1) && (_my_pe() != master))
    for (ii = 0; ii < nlong; ii++)
      if (rbuf[ii] != (ii + 1))
        fprintf(stderr, "CPU: %d Broadcast Error: got %d instead of %d\n",
                _my_pe(), rbuf[ii], ii + 1);

  /* Finalises                         */

  shfree(pSync);
  exit(EXIT_SUCCESS);
}
```

N-to-M Collective Communication Functions with N-to-N default Pattern

‘smatestC3.c’ tests the shmem_collect function

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <mpp/shmem.h>

int main(int argc, char *argv[]) {

  /* General initialisations           */

  int ii, numprocs, count, d, nlong;
  int *sbuf = NULL, *rbuf = NULL, *displ = NULL;
  long *pSync = NULL;

  numprocs = _num_pes();
  nlong = _my_pe() + 1;
```

```
  if (numprocs == 1) {
    fprintf(stderr,"Using more than 1 CPU makes the tests of this program"
           " more interesting\n");
    exit(EXIT_FAILURE);
  }

  displ = malloc(sizeof(int) * numprocs);
  count = 0;
  for (ii = 0; ii < numprocs; ii++) {
    displ[ii] = count;
    count = count + ii + 1;
  }
  pSync = shmalloc(sizeof(long) *_SHMEM_COLLECT_SYNC_SIZE);
  for (ii=0; ii < _SHMEM_COLLECT_SYNC_SIZE; ii++)
    pSync[ii] = _SHMEM_SYNC_VALUE;
  rbuf = shmalloc(sizeof(int) * count);
  for (ii = 0; ii < count; ii++)
    rbuf[ii] = 0;
  sbuf = shmalloc(sizeof(int) * nlong);
  for (ii = 0; ii < nlong; ii++)
    sbuf[ii] = ii;

  shmem_barrier_all();      /* Wait for all CPUs to initialise pSync   */

  /* Collect function                    */

  shmem_collect( rbuf, sbuf, nlong, 0, 0, numprocs, pSync );

  ii = d = 0;
  while (ii < numprocs) {
    for(count = 0; count <= ii; count++)
      if (rbuf[d + count] != count)
        fprintf(stderr, "CPU: %d Collect Error: got %d instead of %d\n",
                _my_pe(), rbuf[d + count], count);
    d = displ[count];
    ii++;
  }

  /* Finalises                           */

  shfree(sbuf);
  shfree(rbuf);
  shfree(pSync);
  free(displ);
  exit(EXIT_SUCCESS);
}
```

N-to-M Collective Communication Functions with N-to-1-to-N default Pattern

‘smatestC2.c’ tests the shmem_fcollect function

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <mpp/shmem.h>

int main(int argc, char *argv[]) {

  /* General initialisations             */
  int ii, numprocs;
  int *sbuf = NULL, *rbuf = NULL;
```

```
  long *pSync = NULL;

  numprocs = _num_pes();

  if (numprocs == 1) {
    fprintf(stderr,"Using more than 1 CPU makes the tests of this program"
           " more interesting\n");
    exit(EXIT_FAILURE);
  }

  sbuf = shmalloc(sizeof(int) * numprocs);
  for (ii = 0; ii < numprocs; ii++)
    sbuf[ii] = ii;
  rbuf = shmalloc(sizeof(int) * numprocs);
  for (ii = 0; ii < numprocs; ii++)
    rbuf[ii] = 0;
  pSync = shmalloc(sizeof(long) *_SHMEM_COLLECT_SYNC_SIZE);
  for (ii=0; ii < _SHMEM_COLLECT_SYNC_SIZE; ii++)
    pSync[ii] = _SHMEM_SYNC_VALUE;

  shmem_barrier_all();      /* Wait for all CPUs to initialise pSync   */

  /* Fcollect function                  */

  if (_my_pe()%2 == 1)
    shmem_fcollect( rbuf, sbuf, numprocs, 1, 1, numprocs/2, pSync );

  if (_my_pe()%2 == 1)
    for (ii = 0; ii < numprocs; ii++)
      if (rbuf[ii] != ii)
        fprintf(stderr, "CPU: %d Fcollect Error: got %d instead of %d\n",
                _my_pe(), rbuf[ii], ii);

  /* Finalises                          */

  shfree(sbuf);
  shfree(rbuf);
  shfree(pSync);
  exit(EXIT_SUCCESS);
}
```

‘smatestC1.c’ tests the shmem_int_sum_to_all

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <mpp/shmem.h>

int main(int argc, char *argv[]) {

  /* General initialisations     */

  int ii, numprocs, nlong;
  int *sbuf = NULL, *rbuf = NULL;
  int *pWrk = NULL;
  long *pSync = NULL;

  numprocs = _num_pes();
  nlong = 10;
```

```
  if (numprocs == 1) {
    fprintf(stderr,"Using more than 1 CPU makes the tests of this program"
           " more interesting\n");
    exit(EXIT_FAILURE);
  }

  sbuf = shmalloc(sizeof(int) * nlong);
  for (ii = 0; ii < nlong; ii++)
    sbuf[ii] = ii;
  rbuf = shmalloc(sizeof(int) * nlong);
  for (ii = 0; ii < nlong; ii++)
    rbuf[ii] = 0;
  pSync = shmalloc(sizeof(long) *_SHMEM_REDUCE_SYNC_SIZE);
  for (ii=0; ii < _SHMEM_REDUCE_SYNC_SIZE; ii++)
    pSync[ii] = _SHMEM_SYNC_VALUE;
  pWrk = shmalloc(sizeof(int) * _SHMEM_REDUCE_MIN_WRKDATA_SIZE);

  shmem_barrier_all();      /* Wait for all CPUs to initialise pSync   */

  /* Reduction function                */

  if (_my_pe()%2 == 1)
    shmem_int_sum_to_all(rbuf, sbuf, nlong, 1, 1,
                         numprocs/2, pWrk, pSync );

  if (_my_pe()%2 == 1)
    for (ii = 0; ii < nlong; ii++)
      if (rbuf[ii] != ii * (numprocs/2))
        fprintf(stderr, "CPU: %d Reduce Error: got %d instead of %d\n",
                _my_pe(), rbuf[ii], ii * (numprocs/2));

  /* Finalises                         */

  shfree(sbuf);
  shfree(rbuf);
  shfree(pSync);
  shfree(pWrk);
  exit(EXIT_SUCCESS);
}
```

Conditional Atomic Communication Functions

‘smatestC5.c’ tests the shmem_int_cswap function

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <mpp/shmem.h>

int main(int argc, char *argv[]) {

  /* General initialisations           */

  int numprocs, cond, master, result, remote;
  int target, value;

  numprocs = _num_pes();
  master = numprocs - 1;
  remote = master/2;
```

```
  if (numprocs == 1) {
    fprintf(stderr,"Using more than 1 CPU makes the tests of this program"
           " more interesting\n");
    exit(EXIT_FAILURE);
  }

  target = value = 0;
  if (_my_pe() == master) {
    value = 7;
    cond = 13;
  }

  if (_my_pe() == remote)
    target = 13;

  /* Cswap function                     */

  shmem_barrier_all();

  if (_my_pe() == master)
    result = shmem_int_cswap(&target, cond, value, remote);

  shmem_barrier_all();

  if ( (_my_pe() == remote) && (target != 7) )
    fprintf(stderr, "CPU: %d Swap Error got %d\n", remote, target);

  if ( (_my_pe() == master) && (result != 13) )
    fprintf(stderr, "CPU: %d Swap Error got %d\n", master, result);

  /* Finalises                          */

  exit(EXIT_SUCCESS);
}
```

Unconditional Atomic Communication Functions

‘smatestC4.c’ tests the shmem_int_swap function

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <mpp/shmem.h>

int main(int argc, char *argv[]) {

  /* General initialisations            */

  int numprocs, master, result, remote;
  int target, value;

  numprocs = _num_pes();
  master = numprocs - 1;
  remote = master/2;

  if (numprocs == 1) {
    fprintf(stderr,"Using more than 1 CPU makes the tests of this program"
           " more interesting\n");
    exit(EXIT_FAILURE);
  }
```

```
  target = value = 0;
  if (_my_pe() == master)
    value = 7;
  if (_my_pe() == remote)
    target = 13;

  /* Swap function                        */

  shmem_barrier_all();

  if (_my_pe() == master)
    result = shmem_int_swap(&target, value, remote);

  shmem_barrier_all();

  if ( (_my_pe() == remote) && (target != 7) )
    fprintf(stderr, "CPU: %d Swap Error got %d\n", remote, target);
  if ( (_my_pe() == master) && (result != 13) )
    fprintf(stderr, "CPU: %d Swap Error got %d\n", master, result);

  /* Finalises                            */

  exit(EXIT_SUCCESS);
}
```

Additional Test Programs

The following additional test programs are available:

‘mpitestC3.f90’	Fortran version of ‘mpitestC3.c’
‘mpitestC5.c’	Tests several MPI functions in one program
‘mpitestC6.c’	Tests several MPI functions in one program using communicators
‘pvmtestC5.c’	Tests several PVM functions in one program
‘smatestC0.f90’	Fortran version of ‘smatestC0.c’
‘smatestC1.f90’	Fortran version of ‘smatestC1.c’
‘smatestC5.f90’	Fortran version of ‘smatestC5.c’
‘smatestC6.c’	Tests shmem_short_cswap function with true condition
‘smatestC7.c’	Tests shmem_short_cswap function with false condition

References

The following overview contains the literature to which is referred and also some additional references used to develop the discussed programs.

Books

[Gropp]
W. Gropp, E. Lusk, A. Skjellum. *Using MPI, portable Parallel programming with the Message-Passing Interface*. The MIT Press, Cambridge, Massachusetts, USA, 1994.

[Cray1]
Cray Research. *Message Passing Toolkit: MPI Programmer's Manual*. Manual no.: SR-2197 1.2. Cray Research, USA, 1998.

[Cray2]
Cray Research. *Message Passing Toolkit: PVM Programmer's Manual*. Manual no.: SR-2196 1.2. Cray Research, USA, 1998.

Articles

[Beguelin1]
A. Beguelin, J. Dongarra, A.Geist, W. Jiang, R. Manchek, V. Sunderam. *PVM 3 User's Guide and Reference Manual*. Oak Ridge National Laboratory, Oak Ridge, Tennessee, USA, 1994.

[Beguelin2]
A. Beguelin, J. Dongarra, A.Geist, W. Jiang, R. Manchek, V. Sunderam. *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networking Parallel Computing*. Oak Ridge National Laboratory, Oak Ridge, Tennessee, USA, 1994.

[Egerer]
G. Egerer. *Programmierung in C*. Lecture notes no.: KFA-ZAM-BHB-0140. Forschungszentrum Jülich, Zentralinstitut für Angewandte Mathematik, Jülich, Germany, 1996.

[Dongarra]
J. Dongarra, S. Huss-Lederman, S. Otto, M. Snir, D. Walker. *MPI: The Complete Reference*. The MIT Press, Cambridge, Massachusetts, USA, 1996.

Software

[`man`]
Several on-line system documentation of the Cray T3E was consulted by using the `man` command. All by Cray Research, USA.

Index

This index gives the page at which the wrapper for a communication function is discussed.

Function	Page
MPI_Allgather	28
MPI_Allgatherv	25
MPI_Allreduce	29
MPI_Alltoall	26
MPI_Alltoallv	27
MPI_Bcast	22
MPI_Gather	24
MPI_Gatherv	24
MPI_Reduce	24
MPI_Reduce_scatter	30
MPI_Scan	27
MPI_Scatter	22
MPI_Scatterv	23
pvm_bcast	31
pvm_gather	33
pvm_mcast	32
pvm_reduce	34
pvm_scatter	33
shmem_*broadcas*t	36
shmem_*collect*	37
shmem_*fcollect*	39
shmem_*type_cfunc*	42
shmem_*type_func*	43
shmem_*type_func*_to_all	40

‘*broadcast*’, ‘*collect*’, ‘*fcollect*’, ‘*cfunc*’ and ‘*func*’ have to be replaced by the concerning function names, whereas ‘*type*’ has to be replaced by the concerning function’s type.